Building an Embedded Control Program Workload

Heng Yu and Grant Weddell

School of Computer Science, University of Waterloo

February 21, 2005

Contents

1 Introduction

An *Embedded control program* (*ECP*) is a software system in which a collection of modules or subsystems access the common database in main memory through a set of pre-defined transaction types [11]. Many legacy systems, *e.g*,control programs of private branch exchanges (PBXs) and operating systems, are good examples of ECPs. Moreover, in the new mobile computing area, the control softwares on handheld computers also fall into the ECP category.

There has been previous research work to provide database management for ECPs [5] [7] [11] [8] [9] [14]. Such efforts aim to ease the software development and maintenance of ECPs with two *DB payoffs*. First, a relational or object data model along with a declarative query language gives a conceptual view of each ECP subsystem, which reduces the development cost. Second, an ECP subsystem inherits automatically from the database engine solutions to concurrency control, reliability, and security issues. Hereinafter, we limit the scope of the term "ECP" to the ECPs with the proposed database wrappers.

In particular, we address to the concurrency control issue for ECPs in [14]. In the part of experiments in [14], we took the OLTP benchmark TPC-C [2], and built up from it the input for our experimental simulations. However, we have noticed the limitations of the TPC-C workload. It is mainly for on-disk database, and does not fit very well for the main-memory data processing in ECPs. Such limitations motivate our work in this technical report. We believe it is important to explore some ECPs in the real world, and make a new workload from them.

We decide to take the operating system MINIX [10] as our study case. A set of system calls that access a set of shared kernel data structures are selected as ECP transaction types. Furthermore, we extract the quantitative elements, *e.g*,costs and probabilities, from the source code. This report shows the current status of our work.

The remainder of the report is organized as such. Section 2 reviews the TPC-C workload in [14] and summaries its limitations. Section 3 introduces our decisions for designing the new workload. Section 4 changes the definition of the transaction type in [14]. Section 5, the main part of the report, gives the detailed steps to obtain the transaction types. Section 6 is about getting the probabilities of the transaction types to finalize the workload. Section 7 addresses to getting the costs of locking. Section 8 concludes the report and suggests the future work.

2 Review of the TPC-C workload

2.1 Definitions of the ECP workload

In [14], the *transaction types* are defined as probabilistic finite state machines, and a transaction is an instance of its transaction type. A set of transaction types form a *transaction system*, which represents a workload.

Definition 2.1 *[14] Let D be a* data set *that consists data items in the database, and each data item is identified with a name.*

A transaction type *T is defined as $T = \langle N,\ s,\ F,\ A,\ data,\ duration,\ prob\rangle$.*

- *N is a set of states, each representing an operation.*
- *$s \in N$ is a starting state.*
- *$F \subseteq N$ is a set of terminating states.*
- *$A \subseteq (N - F) \times N$. A is a set of transition arcs from states to states. Terminating states have no outgoing arcs.*
- *data is a function from N to D. For each $n \in N$, $data(n)$ is the data item accessed at state n.*

- *duration is a function from N to nonnegative real number set. For each $n \in N$, duration(n) is the time cost for state n to access data(n).*
- *prob is a function from A to real numbers between 0 and 1. For each arc $\langle n_1, n_2 \rangle \in A$, prob$(\langle n_1, n_2 \rangle)$ is the probability that a transaction (defined below) goes from n_1 to n_2. For each fixed $n \in (N - F)$,*

$$\sum_{\langle n,n' \rangle \in A} prob(\langle n, n' \rangle) = 1.$$

A transaction *t of transaction type $T = \langle N, s, F, A, data, duration, prob \rangle$ is a sequence n_0, $\cdots$, n_k, where $n_i \in N$ for $0 \le i \le k$. Moreover, $n_0 = s$, $n_k \in F$, and for each adjacent pair n_{i-1}, n_i such that $1 \le i \le k$, $\langle n_{i-1}, n_i \rangle \in A$.*

A transaction system *$S = \langle D, TS, prob \rangle$ is characterized by a set of transaction types $TS = \{T_1$, $\cdots$, $T_k\}$, on a data set D, and a function prob from TS to real numbers between 0 and 1. For each $T_i \in TS$, prob(T_i) specifies the probability that a transaction of the transaction type T is chosen when it is to select the next transaction to execute. $\sum_{T_i \in TS} prob(T_i) = 1$. Moreover, $D = \bigcup_{T_i \in TS} \{data_i(n) \mid n \in N_i\}$, where each $T_i = \langle N_i$, s_i, F_i, A_i, $data_i$, $duration_i$, $prob_i \rangle$.*

2.2 The TPC-C workload

To carry on our experiments to evaluate the concurrency control techniques on ECPs, we need a workload of ECP. Because adding database management to ECPs is a new field of research, we did not have such a workload at hand at the stage of [14]. Therefore, we took the TPC-C benchmark [2] as a transaction system. TPC-C contains 5 transactions that access 9 tables, and each transaction has a probability. We converted each TPC-C transaction to a transaction type under our workload definitions. We first transformed the transaction programs to finite state machines. The steps in [14] are:

1. Mapping each embedded SQL statement (one-table operation or two-table join) in the TPC-C transaction program to a component in the transaction type:
   (a) if it is a **SELECT** or **UPDATE** on a single table A, we add a state n and set *data*(n) to A (Figure 1(a));
   (b) if it is a two-table join on table A and B:
      i. if the two tables are pre-selected on there keys before they are joined, we add two nodes n_1 and n_2 such that *data*$(n_1) = A$ and *data*$(n_2) = B$, and let n_2 be the only successor of n_1 (Figure 1(b));
      ii. otherwise, the join is transformed to Figure 1(c).
2. Transforming control flow in the C programming language to the components:
   (a) a sequential execution of two embedded SQL statement blocks s_1, $\cdots$, s_n is transformed to a lineal path through these nodes as in Figure 2(a);

(b) an **if-else** statement is transformed to a branch out in the graph (Figure 2(b));

(c) a loop statement is transformed as Figure 2(c).

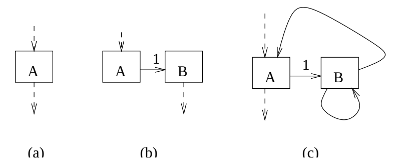

Figure 1: Transformations of queries

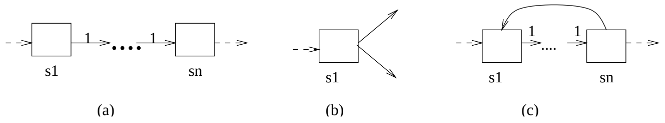

Figure 2: Transformations of control flows

For the join in Case 1(b)ii, the probabilities on the arcs in Figure 1(c) can be estimated from the table cardinality and query selectivity. For **if-else** statements (Case 2b), the probability on each branch in Figure 2(b) are obtainable from the TPC-C specifications. In Case 2c, if it is a **for** loop of the form,

```
for(i=0; i<n; i++){
 S1; ...; Sn;
}
```

we set the probability from s_n back to s_1 to $\frac{n-1}{n}$, and the probability to exit the loop from s_n to $\frac{1}{n}$. If it is a **while** loop, unfortunately, we have to rely on some "guess work" to approximate the probabilities.

Next we partitioned each table to $table_0, \cdots, table_n$, in which n is a *partition factor*. We also added indexes if it is necessary. If the query is a key-matching search, we change the state as in Figure 3(a). If it is a fully sequential table scan, we change it according to Figure 3(b) [14].

The next task in [14] is getting $duration(n)$ for each state n. Following the TPC-C specifications, we created tables and indexes in an IBM DB2 database and inserted row into the tables. For all embedded SQL statements in the TPC-C programs, we ran the SQL statements in DB2 with plugged-in parameters, and got the time costs on the tables and indexes from the query plan graph that are provided by the DB2 visualization tool.

To evaluate the performance of the concurrency control techniques on ECPs, it is important that the workload can be used to simulate main-memory data. To describe the main-memory data and on-disk data situations separately, in [14] we used two parameters, *waiting factor* and *logging factor*. We separate the

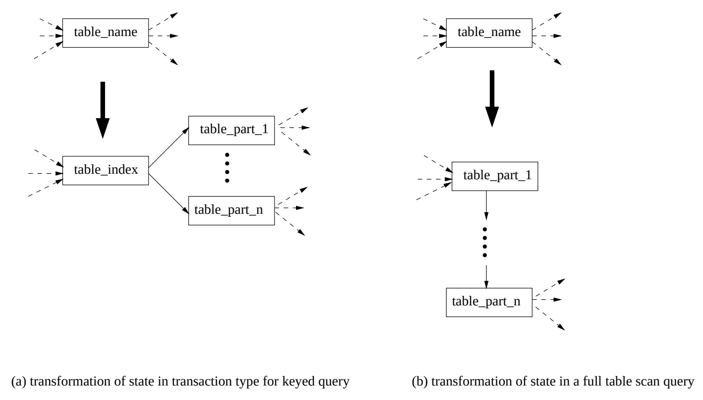
table_name
table_index
table_part_1
table_part_n
(a) transformation of state in transaction type for keyed query
table_name
table_part_1
table_part_n
(b) transformation of state in a full table scan query

Figure 3: Adding indexes and partitions

duration of each state to a *CPU time* and a following *waiting time*. During the CPU time, a transaction occupies the CPU exclusively. The waiting time is for I/O and network delaying, in which the waiting transaction relinquishes the CPU and another transaction can use it. The waiting factor is the ratio of the waiting time to the CPU time. Moreover, for two-phase locking simulation, we also added a *logging duration* for each state that updates data, which also contains a CPU time and a waiting time. The CPU time and waiting time of logging are obtained from those of the original write by multiplying a *logging factor*. For on-disk database, because I/O cost dominates the data processing overhead, and logging can be done efficiently, we set the waiting factor to a large value, *e.g*,10, and set the logging factor to a small value, e.g 0.2. On the contrary, for main-memory data repository, the CPU time is the main factor and the logging overhead is relatively high, we set the waiting factor to a small value, *e.g*,1, and logging factor to a large value, *e.g*,5. Such parameter settings influence the experiment results greatly, as reported in [14].

2.3 Limitations of the TPC-C workload

Although we tried to simulate the main-memory database by setting parameters, there are still some crucial features missing [14]:

- Many (although not all) lock-protected data accesses do not involve I/O at all. Their costs need to be measured in terms of CPU costs only.
- The inter-state cost of the host program instructions cannot be ignored.
- Overhead of locking/unlocking operations should be taken into account. Moreover, the implementation of locking protection influences the locking overhead. When the protected data operations are short and have no I/O involved, short interrupt disabling (on uniprocessor systems) and spinlocks (on SMP systems) are preferred to the costly semaphores [12] [4]. If semaphores are used for transaction

control, under deadlock-free situations, latches are preferred than locks because they have much lower overhead [13].

- To reduce the locking overhead, setting the unit of lock protection to a coarse granularity may be preferable. From the main-memory database experiences [6] [3], the table-level granularity and even database-level granularity (serial execution) are good choices.

As our TPC-C workload is derived from the on-disk IBM DB2 database, it is hard to capture all these features except the last one. We find it is important to design a dedicated workload for ECP applications in addition to the previous TPC-C one. For the time being, we expect to satisfy the features 2.3, 2.3, and 2.3. We describe the CPU costs in terms of number of instructions that process data.

3 Choice of an open-source ECP

If there were a main-memory database which had tools to provide the relevant cost information similar to that in DB2, we could run the TPC-C queries in such a database and obtain new costs, as we did in [14]. However, to the best of our knowledge, no such product has such functionalities as DB2. Consequently, we take an alternative approach. We decide to select an open-source ECP as our study case, and derive our new workload from its source code.

Open-source operating systems are good ECP study cases. The justifications are:

- The kernel data of the operating system are memory-resident.
- The semantics of the kernel data structures and system calls are well-known. Therefore, the former can be re-engineered to relational data, while the latter can be modelled as pre-defined transaction types.
- The operating systems are mainly written in C language. The C-compilers can generate the corresponding low-level assembly instructions. Therefore, we can sum up the numbers of assembly instructions to approximate the costs.

There are two candidates of such operating systems, *Linux* [4] [1] and *MINIX* [10]. Linux is a sophisticated operating system widely used in industry. It has a rich collection of mechanism for mutual exclusion and synchronization, *e.g*,atomic operations, spinlocks, and semaphores. However, the Linux source code is too complicated for our current purpose. The reasons are:

- The programs are written by "hacker programmers". The control flows are often difficult to follow.
- The data structures are huge and messy.
- The implementation of system calls usually involves many levels of function calling, which is not suitable for our workload at this time.

As an operating system for operating system course instruction, MINIX has simple code and data structures, which are fairly readable. The textbook [10] gives a very good explanation of the code. Therefore, it is comparatively easy to manually re-engineer the MINIX system calls to transaction types and measure the costs by the number of CPU instructions. On the other hand, different from Linux, MINIX has no mutual exclusion and synchronization primitives except the basic interrupt disabling. Therefore, when we consider adding the locking overhead, we will use the locking implementations in Linux, or implement it upon MINIX source code by ourselves.

We remind the readers that our goal is to get a workload for ECP transaction processing. Therefore, we do not stick to all the features of MINIX or Linux themselves. MINIX assumes a single user and the system calls are processed sequentially. Early releases of Linux has a non-preemptive kernel, and such limitation is greatly relaxed in the latest Linux 2.6. Although we get our transaction types from the MINIX code and locking overhead from Linux, we just use them as reference and are not restricted to the limitations of the operating systems themselves. Our workload is adjustable from fully preemptive to fully sequential, based on particular experiment settings.

4 Redefinition of transaction types

First, to characterize the inter-state cost in terms of machine instructions, we change the definition of a transaction type in Definition 2.1.

Definition 4.1 *Let D be a* data set *that consists data items in the database, and each data item is identified with a name.*

A transaction type *T is defined as $T = \langle N,\ s,\ F,\ A,\ data,\ cost,\ prob \rangle$.*

- *N, s, F, A, data, and prob are the same as in Definition 2.1;*
- *Change the duration element in Definition 2.1 to the counterpart cost: cost is a function from $N \bigcup A$ to nonnegative integer set. This represents the number of CPU instructions it takes on the $a \in A$ or $n \in N$.*

Other definitions are the same as in Section 2.1.

5 Building transaction types

We build the transaction types from the MINIX code of the system calls. At first, we do not consider the locking implementations and costs. The steps are:

1. Selecting system calls;
2. Modifying and simplifying the C code of the system calls;

3. Mapping the kernel data structures to relational tables;
4. Changing the pure C code to the C code with embedded SQL statements;
5. Generating assembly instructions from the C code at Step 3 and obtaining the costs by summing up the number of instructions;
6. Getting the probabilities for state transitions.

5.1 Selecting system calls

From the MINIX code in [10], we choose 5 system calls, **fork**, **exit**, **waitpid**, **exec**, and **brk** to build up our workload. The reasons that we choose them are:

1. All these system calls are well-known and frequently used in Unix-like operating systems [12] [10];
2. They are highly related to the data structures that represent process and memory chunk descriptors.

5.2 Modifying C code

The MINIX operating system has a microkernel architecture with 4 layers, from the lowest to the highest, *process management*, *I/O tasks*, *server processes*, and *user processes* [10]. The kernel consists of the 2 lowest layers. Many traditional kernel functionalities are moved to the 3rd layer to make the kernel small. In particular, main memory management is put to the *main memory server* (***MM***), and file system is put to the *file system server* (***FS***). Usually several layers are involved to handle a system call.

Example 5.1 *To process the system call **fork**, MM is invoked to assign the memory for the new process, FS is invoked to duplicate the opened file descriptors to the new process, and the low-level kernel is invoked to record the new process for further scheduling. From the code perspective, **MM**, **FS**, and the kernel all have their own **fork** handlers, each implements the corresponding processing in its own module. In addition, all the above three modules have their own process descriptors and process tables. The process descriptor in the kernel keeps the register information of the process, the one in the memory server keeps its memory information, and the one in the file server keeps its opened file information.*

For simplicity, we would like to "flatten" the system call processing to one layer. Because we are mainly interested in the accesses to the process descriptors and the main memory data structures, we focus on the handlers of the system calls in **MM**. We also add some processing from the kernel and **FS**. From the kernel level, we take the part that maintains the ready queue but ignore the processing of the register values. From **FS**, we only take in account the processing of the file descriptors to reflect that the files are duplicated in **fork**, and are closed in **exit** and **exec**. However, the code in **exec** that opens and reads the executable file to get its image, is ignored. Correspondingly, we also combine the process descriptor data structures at different levels. Although in MINIX not all of these data structures and programs are in its kernel part, here we still call them kernel data structures and kernel programs.

We modified the MINIX data structures and programs given in [10]. The modified data structures are listed below, and the modified programs are in Appendix A.1.

Main memory structures MINIX has a very simple memory management. There is no paging and swapping. Each memory chunk assigned to a process is called a *segment*. Each segment has a physical address and a virtual (in-process) address. The length unit of a memory chunks is a *click*, which equals to 256 bytes.

```
typedef unsigned int vir_clicks;
typedef unsigned int phys_bytes;
typedef unsigned int phys_clicks;

struct mem_map_t {
  vir_clicks mem_vir;   /* virtual address */
  phys_clicks mem_phys; /* physical address */
  vir_clicks mem_len;   /* length in clicks */
};
```

The free memory is organized in a list of *holes*. The segments of each process are assigned from the holes whose sizes are big enough.

```
#define NR_HOLES 128   /* system limitation of number of holes */
#define NIL_HOLE (struct hole_t *) 0  /* null pointer */

struct hole_t {
  phys_clicks h_base;    /* physical address */
  phys_clicks h_len;     /* length  */
  struct hole_t *h_next; /* pointer to the next entry */
} hole[NR_HOLES];
```

Opened file descriptor In MINIX, there is a global list which stores the in-memory *inodes* of the opened files.

```
#define NR_INODES 65  /* the maximum number of opened files */

typedef short dev_t;  /* device number */
typedef long time_t;  /* time */
typedef unsigned short ino_t;  /* inode number */

struct inode_t {
  int    count;     /* reference count */
  time_t i_ctime;   /* when was inode itself changed */
  dev_t i_dev;      /* which device is the inode on */
  ino_t i_num;      /* inode number on its device */
} inode[NR_INODES];
```

A *file descriptor* structure **filp_t** associates a process descriptor with an inode structure. It contains a **position** field for the process to read or write the file. The process descriptor maintains an array of pointers to file descriptors of the files the process opens. After a system call **fork**, the new child process shares the same file descriptors with the parent, and the reference count of each shared descriptor is incremented by 1.

```
#define NR_FILPS 128   /* size limit of the filp table */
#define NIL_FILP (struct filp_t *)0

typedef unsigned short mode_t; /* file type and permission bits */
typedef unsighed long off_t;   /* position in a file */

struct filp_t {
  int filp_count;           /* how many file descriptors share this structure*/
  struct inode_t *filp_ino; /* pointer to the inode */
  mode_t mode;              /* file mode */
  off_t  position;          /* position in the file for read and write */
} filp[NR_FILPS];
```

Process descriptor The definition of the process descriptor is presented below.

```
#define NR_PROCS 32   /* maximum number of processes */

#define IN_USE 001    /* the process array element is used */
#define WAITING 002   /* the process is waiting: it has called waitpid
                         and is waiting a child to exit */
#define HANGING 004   /* the process is hanging: it has called exit and
                         is waiting for the parent to call waitpid */
#define SEPARATE 040  /* the process has separate text and data segments */

#define NR_SEGS 3     /* text, data, and stack segments */
#define T       0     /* text */
#define D       1     /* data */
#define S       2     /* stack */

#define OPEN_MAX 20   /* maximum number of files opened by a process */

typedef unsigned reg_t; /* the register value format */

struct proc_t {
  pid_t  pid;         /* process id */
  pid_t  parent_pid;  /* id of parent process */
  pid_t  wpid;        /* pid this process is waiting for */

  char   exitstatus;  /* storage for status when process exits */
  char   sigstatus;   /* storage for signal number for killed processes */

  struct mem_map_t seg[NR_SEGS]; /* points to text, data, stack segments */

  /* File identification for sharing */
  ino_t   ino;    /* inode number of file */
  dev_t   dev;    /* device number of file system */
  time_t  ctime;  /* inode changed time */

  unsigned proc_flags; /* flag bits: IN_USE, WAITING, HANGING, SEPARATE*/
  reg_t sp;          /* stack pointer */

  struct filp_t *fp_filp[OPEN_MAX];  /* opened file descriptors */
  struct proc_t *p_next_ready;  /* pointer in the ready queue */
} proc[NR_PROCS];
```

Most of the fields of the process descriptor are from the process descriptor in **MM**. The **fp_filp** field is from the descriptor in **FS**. The **p_next_ready** field is from the MINIX kernel, and it is the pointer to the next process in the ready queue for scheduling.

```
#define NQ 3
#define TASK_Q   0   /* task */
#define SERVER_Q 1   /* server */
#define USER_Q   2   /* user process */

struct proc_t *rdy_head[NQ];  /* head of a queue */
struct proc_t *rdy_tail[NQ];  /* tail of a queue */
```

Each process has a unique identifier. Some identifiers are assigned by MINIX for system processes. Here are the 2 of our interest, the MM server process and the *init* process.

```
#define MM_PROC_NR 0   /* process number of memory manager */
#define INIT_PID 1     /* init process */
```

5.3 Mapping data structures to relational tables

We map the kernel data structures defined in Section 5.2 to relational tables. Most of the mappings are straightforward. They are listed below, in which the left hand side is data structure in the C language, and right hand side is the table name.

- **proc[NR_PROCS]** → **proc**;
- **inode[NR_INODES]** → **inode**;
- **filp[NR_FILPS]** → **filp**;
- **hole[NR_HOLES]** → **hole**;
- **rdy_head[USER_Q]**, **rdy_tail[USER_Q]** → **ready_user_proc**;
- **fp_filp[OPEN_MAX]** in **proc[NR_PROCS]** → **proc_filp**;
- **seg[NR_SEGS]** in **proc[NR_PROCS]** → **segment**.
- **1..30000** → **pids**

In the data structure definitions in Section 5.2, the segments (**seg**) are defined as a nested array inside the process descriptor (**proc_t**). Since the relational model supports only "flat" tables, we put the segment information into a separate table **segment**, and define an artificial identifier field **seg_id** for **segment** and the corresponding foreign keys in the table **proc**. Moreover, the **fp_filp** array in **proc_t** shows the $m : n$ relationship between process descriptors and file descriptors, which reflects that several files can shared the same file descriptor and a process can open several files. This cannot be encoded in relational model without an auxiliary table. Therefore, we introduce a key **filp_id** in **filp** and create an additional table **proc_filp** which contains the foreign keys to both **proc** and **filp**.

The DDL statements that create the tables are listed below.

segment table of main memory segments that hold code, data, or stack of processes:

```
create table segment(
  seg_id  integer NOT NULL,
  virtual_address integer,
  physical_address integer,
  length integer,
  PRIMARY KEY (seg_id)
)
```

proc process descriptor table:

```
create table proc(
  pid  integer NOT NULL,
  parent_pid integer,
  wait_pid integer,
  exit_status integer,
  sig_status integer,
  data_segment integer,
  text_segment integer,
  stack_segment integer,
  ino    integer,
  dev    integer,
  ctime  timestamp,
  stack_pointer integer,
  proc_flags integer,
  PRIMARY KEY (pid),
  FOREIGN KEY (parent_pid) REFERENCES   proc(pid),
  FOREIGN KEY (wait_pid)   REFERENCES   proc(pid),
  FOREIGN KEY (data_segment) REFERENCES segment(seg_id),
  FOREIGN KEY (text_segment) REFERENCES segment(seg_id),
  FOREIGN KEY (stack_segment) REFERENCES segment(seg_id)
)
```

In the table definition, **proc_flags** is an integer whose bits correspond to the flags **SEPARATE**, **WAITING**, and **HANGING**, which are defined together with the process descriptor **proc_t** in Section 5.2. Another flag **IN_USE** defined there does not need to be mapped, as it is shown by whether the row "exists" in the table.

inode table of the in-memory inodes of opened files.

```
create table inode (
  inode integer,
  dev   integer,
  ctime timestamp,
  count integer,
  PRIMARY KEY (inode)
)
```

filp file descriptor table:

```
create table filp(
  filp_id  integer NOT NULL,
  inode_id integer,
  position integer,
  mode     integer,
  filp_count integer,
  PRIMARY KEY (filp_id)
)
```

proc_filp table that associates processes to their file descriptors:

```
create table proc_filp(
  proc_id  integer,
  flip_id  integer,
  FOREIGN KEY (proc_id) REFERENCES proc(pid),
  FOREIGN KEY (proc_id) REFERENCES filp(filp_id)
)
```

hole free memory hole table:

```
create table hole (
  base integer,
  len  integer
)
```

pid table of integers from 1 to 30000 to get a process identifier:

```
create table pid (
  pid  integer
)
```

ready_user_proc table of user processes that are ready to run:

```
create table ready_user_proc (
  pid integer,
  FOREIGN KEY (pid) REFERENCES proc(pid)
)
```

5.4 Changing the pure C code to the C code with embedded SQL statements

Following the mapping from C-language data structures to relations in Section 5.3, we also transfer the pure C program in Appendix A.1 to the corresponding C program with embedded SQL statements (*C/SQL*). The main work at the step is changing the code blocks that operate on the kernel data structures to embedded SQL statements that manipulate the mapped tables. The C/SQL programs for the system calls are listed in Appendix A.2.

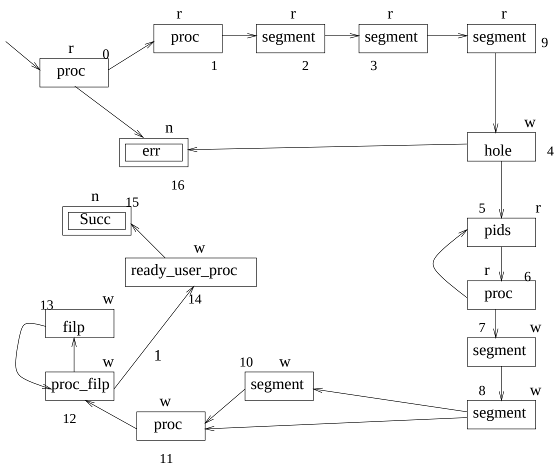

Figure 4: The finite state machine of **fork**

5.5 Obtaining finite state machines from the C/SQL code

Given the C/SQL programs of the transaction types, we can follow the methods in [14] (also reviewed in Section 2.2 of this report) to transform them to finite state machines. However, we do not partition the table, because we assume that the unit of protection for main memory databases should be of coarse granularity, *i.e*,the table level. The finite state machines are given in Figure 4 to 8. The numbers attached to states are state identifiers. In addition, the **r** or **w** attached to each state indicates whether the state read or write the data item.

5.6 Generating assembly instructions and getting the costs

The next task is getting the costs at each arc and each state in the finite state machine to build up the final transaction types. Because we map the statements in the C program to the embedded SQL statements, we can view the C statements as the low-level query plan for the SQL statements. Therefore, if we can find the cost of a "block" of C statements that represents a query plan, we can then get the costs of the states involved in the SQL query. Moreover, from the C statements that do not belong to any block corresponding to a query plan, we get the inter-state costs on the arcs.

Currently we measure the cost of a sequence of C statements with the total number of their assembly instructions. We run the **gcc -g -S** command on the C programs. This produces the assembly code of the program without code optimization. The sequence of assembly instructions for each C statement are generated by the compiler. Therefore, we can get the costs at the states and arcs by summing up the number of instructions of their statements. When there are loops or conditional statements in the code, we have to approximate the numbers by *ad hoc* "averaging" or even "guessing".

We characterize the general features of the kernel data structures in Section 5.2 as below:

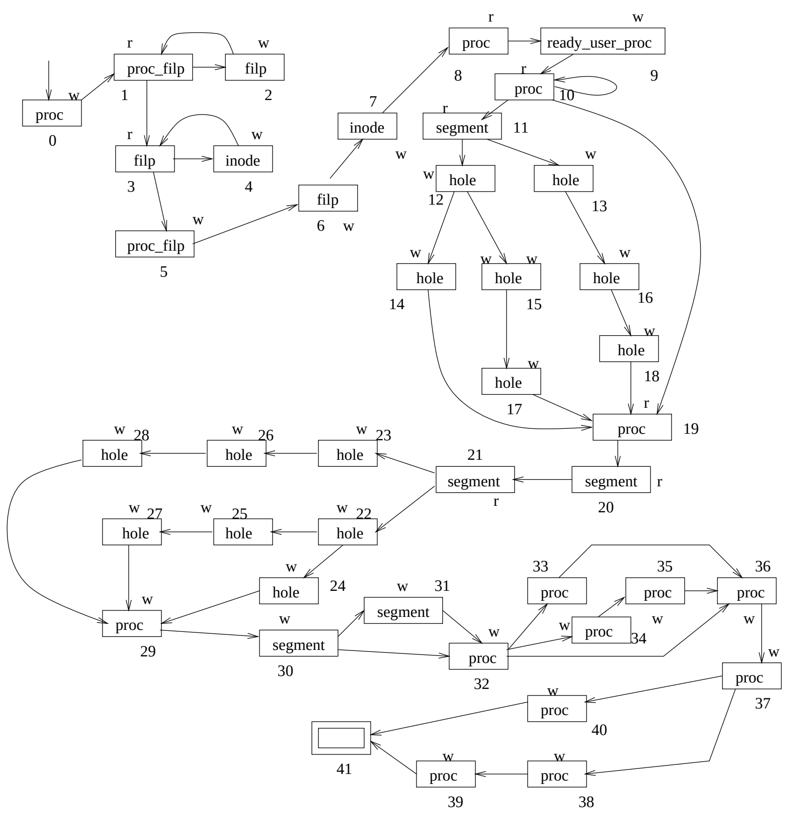

Figure 5: The finite state machine of **exit**

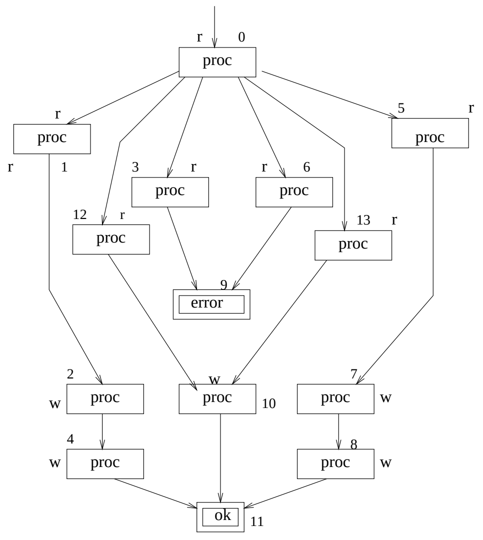

Figure 6: The finite state machine of **waitpid**

hole: **hole** is an array of **hole_t** elements of size 128, which specifies the maximal number of holes. However, the real hole list is chained by the **h_next** pointer field in **hole_t**. We assume that on average the hole list has $\frac{128}{2} = 64$ elements. For segment allocation, we assume on average the searching goes over $\frac{64}{2} = 32$ elements before it finds a hole big enough for the segment.

proc: **proc** is an array of **proc_t** elements of size 32. Whether the element is a valid is indicated by the lowest bit (**IN_USE**) in the **proc_flags** field of **proc_t**. We assume that $\frac{3}{4} \times 32 = 24$ elements in the array are in use. Moreover, we also assume:

- Given a process id, we assume that the searching scans $\frac{32}{2} = 16$ elements in the array to find the corresponding process descriptor.
- On average there are $32 - 24 = 8$ **proc_t** elements are not in use. They are distributed uniformly in the array. To find the first free element in the **proc** array, the searching scans on average $\frac{32}{8} = 4$ elements.

rdy_head: We assume the probability that the ready queue of user processes (**rdy_head[USER_Q]**) is empty is $\frac{1}{10}$, and the respective probabilities that the ready queue of kernel tasks (**rdy_head[TASK_Q]**) and servers (**rdy_head[SERVER_Q]**) are $\frac{1}{4}$.

filp: The descriptors of the files opened of a process is represented by the **fp_filp** array enclosed in the **proc_t**. Each element of the array is a pointer to **filp_t**, and the size of the array is 20. We assume that on average a process open $\frac{20}{2} = 10$ files, so 10 pointers in the array are not NULL.

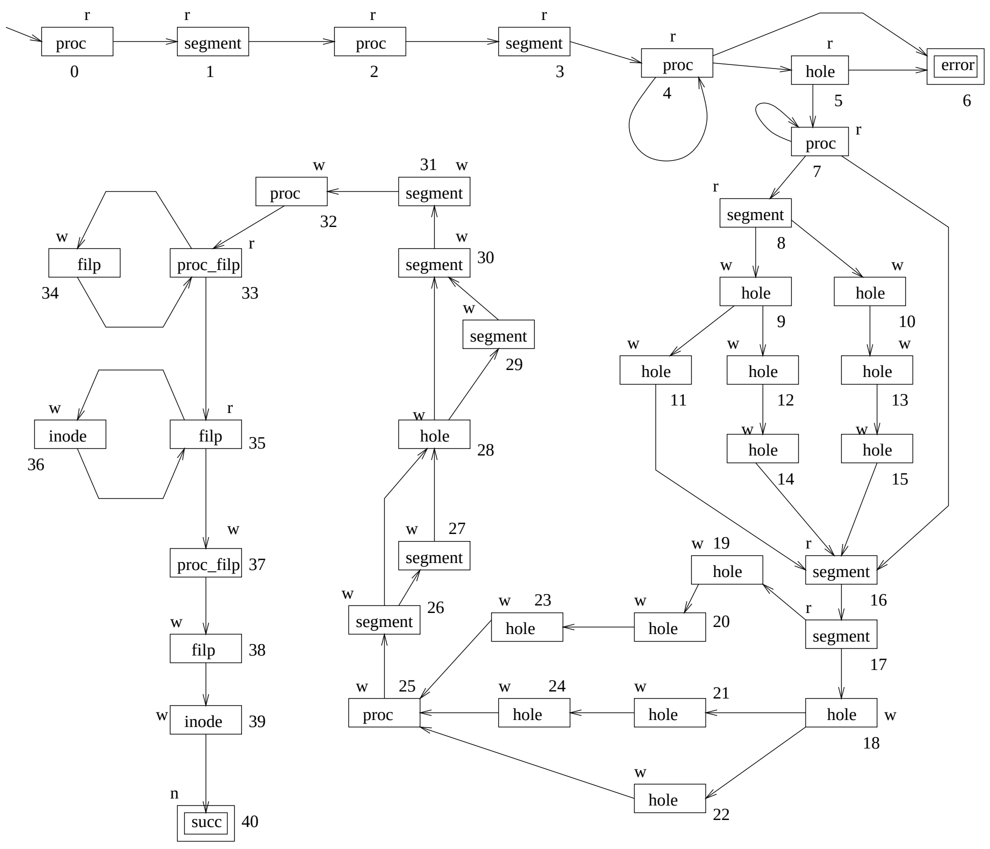

Figure 7: The finite state machine of **exec**

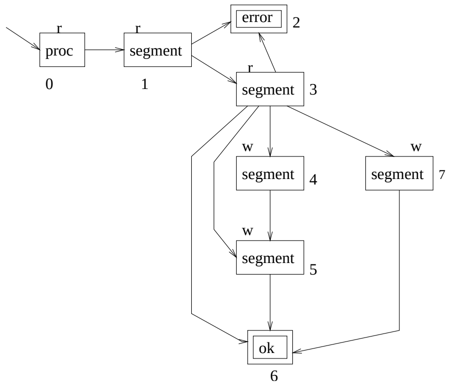

Figure 8: The finite state machine of **brk**

Currently we get the costs manually. Because there are too many technical details in the procedure, we illustrate how to get the costs by examples.

Example 5.2 *In **fork**, we need to get the information of the parent process (state* 1 *in Figure 4). The SQL statement embedded in the C/SQL program (Appendix A.2.1) is:*

```
EXEC SQL SELECT
  parent_pid into :parent_pid,
  wait_pid into :wait_pid,
  exit_status into :exit_status,
  sig_status into :sig_status,
  data_segment into :data_segment,
  text_segment into :text_segment,
  stack_segment into :stack_segment,
  proc_flags into :proc_flags,
  ino into :ino,
  dev into :dev,
  ctime into :ctime
FROM proc
WHERE pid = :current_pid;
```

The C statements that process the query, taken from Appendix A.1.3, are listed with line numbers:

```
/*line 1*/   for(i=0; i<NR_PROCS; i++){
/*line 2*/     if((proc[i].proc_flags & IN_USE != 0) &&
/*line 3*/        (proc[i].pid == parent_pid))
/*line 4*/     {
/*line 5*/       parent_proc = proc[i];
/*line 6*/       break;
/*line 7*/     }
/*line 8*/   }
/*line 9*/   ino = parent_proc.ino;
/*line 10*/  dev = parent_proc.dev;
/*line 11*/  ctime = parent_proc.ctime;
/*line 12*/  sp = parent_proc.sp;
/*line 13*/  wpid = parent_proc.wpid;
/*line 14*/  proc_flags = parent_proc.proc_flags;
```

The corresponding assembly instructions for the above C code are shown below. They are annotated with the line numbers in the C program.

```
!*********************************
!*********** line 1****************
!*********************************
        movl    $0, -12(%ebp)
!1 instruction
!*********************************
.L8:
        cmpl    $31, -12(%ebp)
        jle     .L11
!2 instructions
!*********************************
        jmp     .L9
!1 instruction
!*********************************
!************line 2 ***************
```

```
!***********************************
.L11:
        movl    -12(%ebp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    %eax
        addl    %edx, %eax
        sall    $3, %eax
        movl    proc+60(%eax), %eax
        andl    $1, %eax
        testl   %eax, %eax
        je      .L10
!11 instructions
!***********************************
!************line 3 ***************
!***********************************
        movl    -12(%ebp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    %eax
        addl    %edx, %eax
        sall    $3, %eax
        movl    proc(%eax), %eax
        cmpl    8(%ebp), %eax
        jne     .L10
!10 instructions
!***********************************
!************line 5****************
!***********************************
        movl    -12(%ebp), %edx
        movl    %edx, %eax
        sall    $3, %eax
        addl    %edx, %eax
        sall    %eax
        addl    %edx, %eax
        sall    $3, %eax
        leal    -200(%ebp), %edi
        leal    proc(%eax), %esi
        cld
        movl    $38, %eax
        movl    %eax, %ecx
        rep
        movsl
!14 instructions
!***********************************
!************line 6****************
!***********************************
        jmp     .L9
!1 instruction
!***********************************
!************line 8****************
!***********************************
.L10:
        leal    -12(%ebp), %eax
        incl    (%eax)
        jmp     .L8
!3 instructions
!**********************************
!**********************************
!************line 9****************
.L9:
        movl    -148(%ebp), %eax
        movw    %ax, -222(%ebp)
!2 instructions
!**********************************
!************line 10***************
!**********************************
        movw    -146(%ebp), %ax
        movw    %ax, -224(%ebp)
!2 instructions
!**********************************
!************line 11***************
!**********************************
        movl    -144(%ebp), %eax
        movl    %eax, -228(%ebp)
!2 instructions
!**********************************
!************line 12***************
!**********************************
        movl    -136(%ebp), %eax
        movl    %eax, -240(%ebp)
!2 instructions
!**********************************
!************line 13***************
!**********************************
        movl    -192(%ebp), %eax
        movl    %eax, -236(%ebp)
!2 instructions
!**********************************
!************line 14***************
!**********************************
        movl    -140(%ebp), %eax
        movl    %eax, -232(%ebp)
!2 instructions
!**********************************
```

NR_PROC *is* 32. *By assumption,* $\frac{3}{4}$ *of the elements in* ***proc*** *are in use and they are distributed evenly. In addition, the loop iterates* $\frac{32}{2} = 16$ *iterates on average to find the qualified element that matches the process id parameter. Therefore, from the the number of assembly instructions for each each line of the C* *code, the cost to process the SQL statement is*

$$\begin{aligned}
cost = & 1 + 2 \times 16 && \textit{(line 1)}\\
& + 11 \times 16 && \textit{(line 2)}\\
& + 10 \times 16 \times \frac{3}{4} && \textit{(line 3)}\\
& + 14 && \textit{(line 5)}\\
& + 1 && \textit{(line 6)}\\
& + 3 \times (16 - 1) && \textit{(line 8)}\\
& + 2 + 2 + 2 + 2 + 2 + 2 && \textit{(line 9-14)}\\
= & 401
\end{aligned}$$

Example 5.3 *In **exit**, the opened files are to be closed. The C/SQL statements (from Appendix A.2.2) are:*

```
EXEC SQL UPDATE filp set filp_count = filp_count - 1
WHERE EXISTS (
  SELECT * FROM proc_filp
  WHERE proc_filp.proc_id = :current_pid
  AND   proc_filp.filp_id = filp.filp_id
);

EXEC SQL UPDATE inode set count = count - 1
WHERE EXISTS (
  SELECT * from filp
  WHERE filp.filp_count = 0
  AND filp.inode_id = inode.inode
)

EXEC SQL DELETE FROM proc_filp
WHERE proc_id = :current_pid;

EXEC SQL DELETE FROM filp
WHERE filp_count = 0;

EXEC  SQL DELETE FROM inode
WHERE count = 0;
```

They represent the subgraph of the finite state in Figure 5 generated by Node 1 *to* 7*. We show the subgraph in Figure 9.*

The C program that process the queries, taken from Appendix A.2.2, are:

```
/*line 1*/   for(i=0; i<OPEN_MAX; i++){
/*line 2*/     if((current_proc_ptr->fp_filp[i])!=NIL_FILP){
/*line 3*/       (current_proc_ptr->fp_filp[i])->filp_count--;
/*line 4*/       if((current_proc_ptr->fp_filp[i])->filp_count == 0){
/*line 5*/         (current_proc_ptr->fp_filp[i])->filp_ino->count--;
/*line 6*/       }
/*line 7*/       current_proc_ptr->fp_filp[i] = NIL_FILP;
/*line 8*/     }
/*line 9*/   }
```

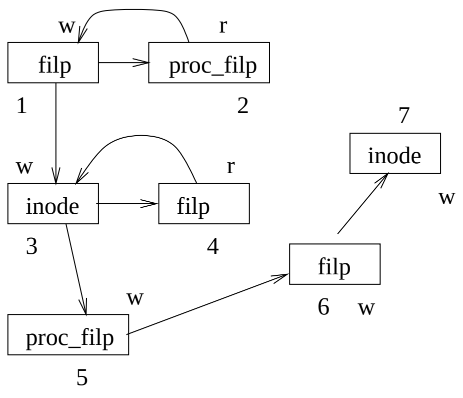

Figure 9: Part of the finite state machine of **exit**

*To map the C statements to the embedded SQL statements, we partition the **for** loop. The first embedded SQL statement that joins **filp** and **proc_filp** is processed by scanning the **fp_filp** array in the process descriptor. So the first join (the loop with with state* 1 *and* 2 *in Figure 9) is processed by line* 1, 2, 3, 8, *and* 9. *The second one that joins **filp** and **inode** is processed by line* 1, 2, 5, 6, 8, *and* 9. *The third deletes the relevant rows in the **proc_filp** table. This is processed by line* 1, 2, 7, 8, 9. *The last two **delete** statements actually have nothing to do at the physical level, because when the reference counts in **inode_t** and **filp_t** equal to* 0, *the entry is free.*

The assembly code is:

```
!**********************************
!************line 1****************
!**********************************
        movl    $0, -4(%ebp)
!1 instruction
!**********************************
.L7:
        cmpl    $19, -4(%ebp)
        jle     .L10
!2 instructions
!**********************************
        jmp     .L8
!1 instruction
!**********************************
!************line 2****************
!**********************************
.L10:
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        cmpl    $0, 68(%edx,%eax,4)
        je      .L9
!4 instructions
!**********************************
!************line 3****************
```

```
!**********************************
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    68(%edx,%eax,4), %eax
        decl    (%eax)
!4 instructions
!**********************************
!************line 4****************
!**********************************
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    68(%edx,%eax,4), %eax
        cmpl    $0, (%eax)
        jne     .L12
!5 instructions
!**********************************
!************line 5****************
!**********************************
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    68(%edx,%eax,4), %eax
        movl    4(%eax), %eax
        decl    (%eax)
!5 instructions
!**********************************
!************line 7****************
!**********************************
.L12:
        movl    -12(%ebp), %edx
        movl    -4(%ebp), %eax
        movl    $0, 68(%edx,%eax,4)
!3 instructions
!**********************************
!************line 9****************
!**********************************
.L9:
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L7
!3 instructions
!**********************************
```

The costs are:

State 1 *is processed by the 2 instructions under **.L1** for line* 1*. Its cost is* 2*.*

State 2 *is processed by line* 2 *and* 3*. As we assume half of the elements in **fp_filp** are non-null pointers and the other half are null, there is* 50% *probability that line* 3 *is executed. Therefore,*

$$\begin{aligned} cost =& 4 && \textit{(line 2)} \\ & + 4 \times 0.5 && \textit{(line 3)} \\ =& 6 \end{aligned}$$

Arc(1, 2) *has cost* 0.

Arc(2, 1) *is done by line* 9. *Its cost is* 3.

State 3 *is similar to state* 1. *Its cost is* 2.

Arc(1, 3) *is processed by* 2 *instructions for line* 1 *(the first one that initializes and last one that jumps). Its cost is* 2.

Arc(3, 4) *has cost* 0.

State 4 *is processed by line* 2, 5, *and* 6 *(empty). As we assume that half of* ***fp_filp*** *elements are not null, the probability that line* 4 *is executed is* 50%. *We also assume the probability that line* 5 *is executed is* 40%.

$$\begin{aligned} cost =& 4 && \textit{(line 2)} \\ &+ 5 \times 0.5 && \textit{(line 4)} \\ &+ 5 \times 0.4 && \textit{(line 5)} \\ =& 9 \end{aligned}$$

Arc(4, 3) *is processed by line* 9. *The cost is* 3.

State 5 *is processed by line* 1, 2, 7, 8 *(empty), and* 9. *As the state occupies the whole loop block,* ***OPEN_MAX*** *is* 20, *and we assume* 10 *elements are not null, the total cost is,*

$$\begin{aligned} cost =& 1 + 2 \times 20 + 1 && \textit{(line 2)} \\ &+ 4 \times 20 && \textit{(line 2)} \\ &+ 3 \times 10 && \textit{(line 7)} \\ &+ 3 \times 19 && \textit{(line 9)} \\ =& 209 \end{aligned}$$

Arc(3, 5) *is done by the last instruction for line* 1. *So the cost is* 1.

Arc(5, 6), State 6 **and State** 7 *are have the cost* 0.

The transaction types with costs we acquired are shown in Figure 10 to 14. The numbers attached to the states and arcs are their costs.

5.7 Getting the probabilities for state transitions

The last step in getting the transaction types is getting the probabilities for the state transitions from states with multiple successors. Some probabilities can be estimated from the number of iterations of the **for** loop, and the others have to be drawn from assumptions.

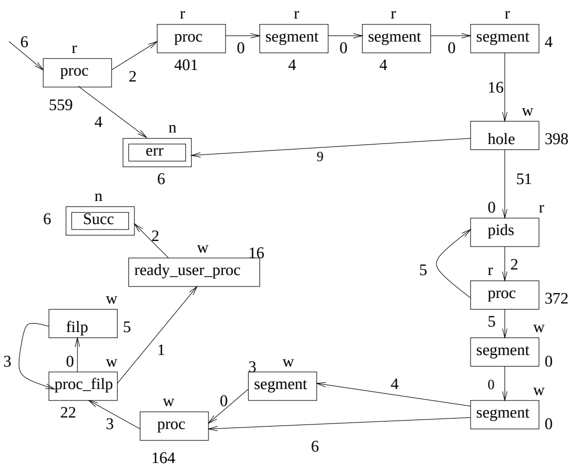

Figure 10: The costs of **fork**

arc	prob	arc	prob	arc	prob
$(0, 1)$	0.95	$(0, 16)$	0.05	$(4, 5)$	0.95
$(4, 16)$	0.05	$(6, 5)$	0.967	$(6, 7)$	0.033
$(8, 11)$	0.67	$(8, 10)$	0.33	$(12, 13)$	0.95
$(12, 14)$	0.05				

Table 1: The probabilities of state transitions in **fork**

Example 5.4 *For the part of* ***exit*** *in Figure 9, as the* ***for*** *loop iterates* 20 *times, we can estimate that the probability from state* 1 *to* 2 *is* $1 - \frac{1}{20} = 0.95$*, and from* 1 *to* 3 *is* 0.05*. For the same reason, the probability from state* 3 *to* 4 *is* 0.95*, and from* 3 *to* 5 *is* 0.05*.*

We include in Table 1 to 5 the probabilities of transitions for the states with multiple successors. The state identifiers in the tables are from Figure 4 to 8.

6 Getting the probability of the transaction types

After we have the set of transaction types, we also need their probabilities to make the final transaction system. Here we assume that there is the same percentage of **fork** and **exit** in the transaction mixture, 80% of forked processes calls **exec**, and 80% of the parent processes call **waitpid** for their child processes. **brk** is the underlying system call that implements **malloc** and **free**. We also assume on average each process calls **brk** for 5 times. So we can calculate the percentage of each transaction type. The probabilities are given in Table 6.

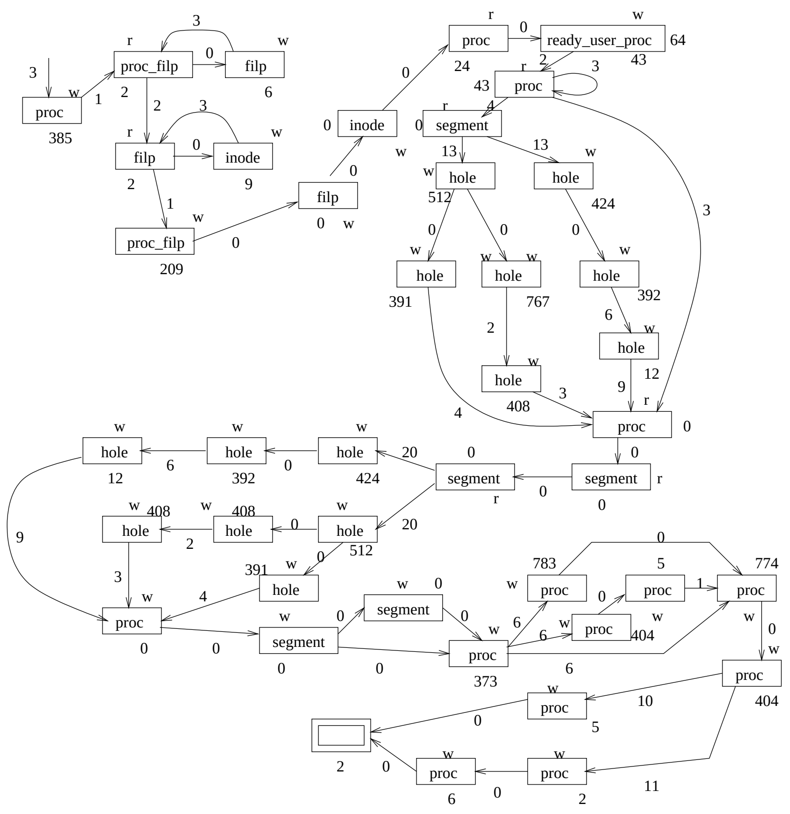

Figure 11: The costs of **exit**

arc	prob	arc	prob	arc	prob
$(1,2)$	0.95	$(1,3)$	0.05	$(3,4)$	0.95
$(3,5)$	0.05	$(10,10)$	0.9375	$(10,11)$	0.0417
$(10,19)$	0.0208	$(11,12)$	0.2	$(11,13)$	0.8
$(12,14)$	0.8	$(12,15)$	0.2	$(21,22)$	0.2
$(21,23)$	0.8	$(22,24)$	0.8	$(22,25)$	0.2
$(30,31)$	0.67	$(30,32)$	0.33	$(32,36)$	0.5
$(32,34)$	0.125	$(32,33)$	0.375	$(37,38)$	0.5
$(37,40)$	0.5				

Table 2: The probabilities of state transitions in **exit**

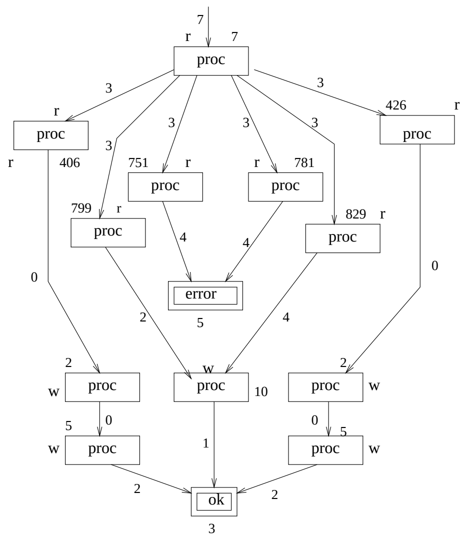

Figure 12: The costs of **waitpid**

arc	prob	arc	prob	arc	prob
$(0,1)$	0.225	$(0,3)$	0.05	$(0,12)$	0.225
$(0,5)$	0.225	$(0,6)$	0.05	$(0,13)$	0.225

Table 3: The probabilities of state transitions in **waitpid**

arc	prob	arc	prob	arc	prob
$(4,4)$	0.9375	$(4,5)$	0.0575	$(4,6)$	0.005
$(5,6)$	0.005	$(5,7)$	0.995	$(7,7)$	0.9375
$(7,8)$	0.0417	$(7,16)$	0.0208	$(8,9)$	0.2
$(8,10)$	0.8	$(9,11)$	0.8	$(9,12)$	0.2
$(17,18)$	0.2	$(17,19)$	0.8	$(18,21)$	0.2
$(18,22)$	0.8	$(26,27)$	0.67	$(26,28)$	0.33
$(28,29)$	0.67	$(28,30)$	0.33	$(33,34)$	0.95
$(35,36)$	0.95	$(35,37)$	0.05		

Table 4: The probabilities of state transitions in **exec**

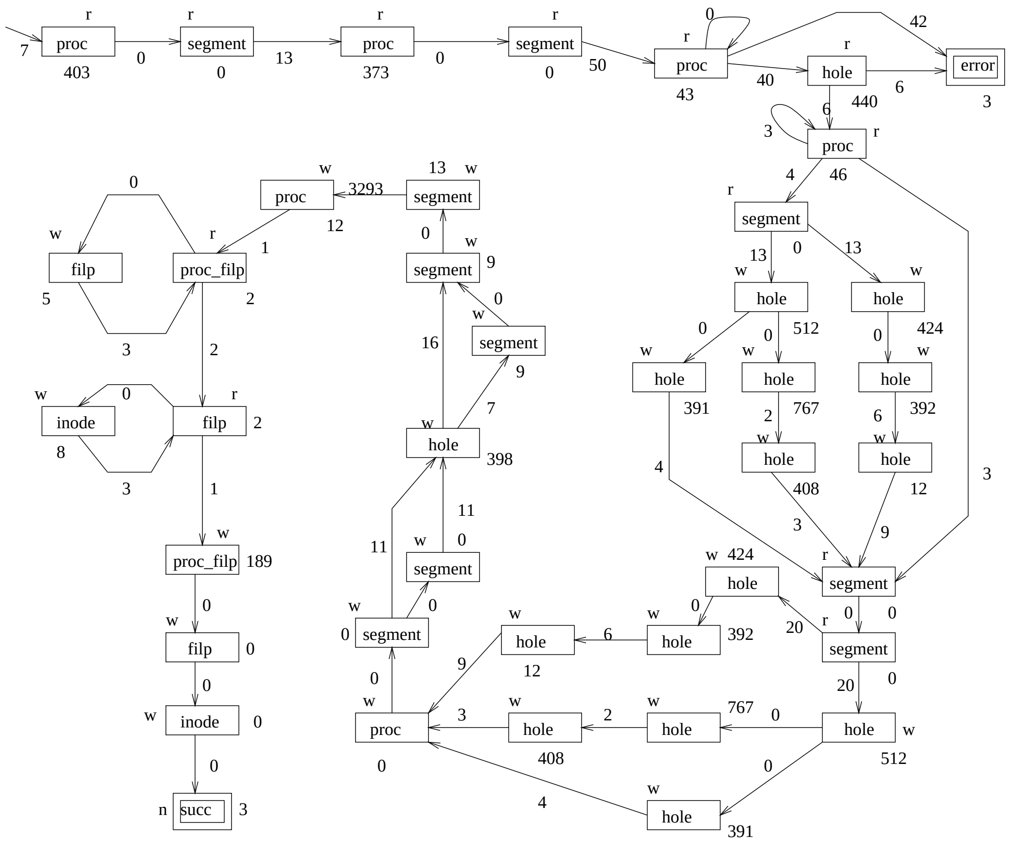
Figure 13: The costs of **exec**

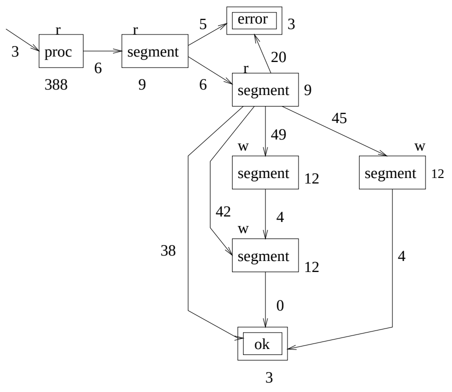
Figure 14: The costs of **brk**

arc	prob	arc	prob	arc	prob
$(1, 2)$	0.05	$(1, 3)$	0.95	$(3, 2)$	0.05
$(3, 4)$	0.095	$(3, 7)$	0.19	$(3, 5)$	0.19
$(3, 6)$	0.475				

Table 5: The probabilities of state transitions in **brk**

transaction type	probability
fork	11.7%
exit	11.7%
waitpid	9.3%
exec	9.3%
brk	58%

Table 6: The probabilities of the transaction types

Hence we finalize the workload.

7 Calculating the costs of locking

To model the workload for main-memory data, we also consider the costs of locking protections. The MINIX system, which is used for eduction, has only interrupt disabling protection. This is equivalent to serial scheduling on a uniprocessor system, and can implemented by one **cli** instruction for locking and one **sti** for unlocking. Additionally, we take into account *spinlock* which is used in UNIX/Linux for protection on SMP [12] [4]. We take the code that implements spinlock from the Linux source code [4]. In Linux, there are two versions of spinlock, exclusive and read-write. Next we analyze their costs separately.

7.1 Exclusive spinlock

For exclusive spinlock, assuming the lock is encoded in a byte **slp**, the assembly code for locking is [4]:

```
1:  lock; decb slp
    jns 3f
2:  cmpb $0, slp
    pause
    jle 2b
    jmp 1b
3:
```

The unlocking code is:

```
lock; movb $1, slp
```

So when the value of **slp** is 1, the lock is free. The locking code will directly jump to label 3. So it takes 2 instructions (the first 2 instructions). Otherwise, the code loops after label 2 and tests whether the value is reset to 1. If it is, the code goes back to label 1 to retest whether the lock is still free. So the resumption of the process takes 4 instructions, including a **jle** and a **jmp**. The unlocking has only 1 instruction.

7.2 Read-write spinlock

The read-write spinlock addresses to the classical reader and writer problem, and allows multiple simultaneous readers and only one write at one time. However, different from high-level semaphore lock, it does not prevent a waiting writer from being starved by the subsequent readers. It is encoded as a long integer value (suppose at address **rwlp**). When the value is set to $0x01000000$, there is no read or write lock holder, and the 1 at the 24th bit shows that the write lock is free. When the value is a positive number, it is the 2's complement of the counter of readers. When it is 0 or negative, there is a writer. The code to acquire a read lock is [4]:

```
        mobl $rwlp, %eax
        lock; subl $1, (%eax)
        jns 1f
        call __read_lock_failed
    1:

     __read_lock_failed:
        lock; incl (%eax)
    1:  cmpl $1, (%eax)
        js  1b
        lock; decl (%eax)
        js __read_lock_failed
        ret
```

A successful locking takes 3 instructions and goes to label 1. So it takes 3 instructions. It takes a waiting processing 4 instructions to acquire the lock after the spinlock holder releases it. They including a **js**, a **decl**, another **js**, and a **ret** in the function **__read_lock_failed**.

The unlocking code is [4]:

```
        lock; incl rwlp
```

The unlocking takes 1 instruction.

The locking procedure for a write request is [4]:

```
        movl $rwlp, %eax
        lock; subl $0x01000000, (%eax)
        jz 1f
        call __write_lock_failed
    1:

    __write_lock_failed:
        lock; add $0x0100000, (%eax)
    1:  cmpl $0x01000000, (%eax)
        jne  1b
        lock; subl $0x0100000, (%eax)
        jnz __write_lock_failed
        ret
```

Similar to the read locking, the write locking takes 3 instructions. And a process that resumes from spinning takes 4 instructions to get the lock. The unlocking takes 1 instruction, as shown below [4].

```
lock; add $0x01000000, rwlp
```

7.3 Locks based on semaphores

In addition, Unix/Linux also uses high-level *semaphores* for synchronization and protection. A lock or latch is usually implemented by a semaphore. We treat exclusive locks (*x-lock*) and locks that allowing sharing (*rw-lock*) separately. Both types of semaphores are implemented in Linux. However, we do not the Linux implementation code because it is too complicated. Instead we implement a simple version of both types of locks based on MINIX data structures, except that the spinlock functions (**spin_lock** and **spin_unlock**) called are from the Linux code in [4].

7.3.1 X-lock

First we consider the costs of x-locks. The data structures of an x-lock are:

```
#define LOCK_FREE 0
#define LOCK_X    1

typedef int lock_mode_t;

/* the waiting queue */
struct wait_queue_t {
  struct proc_t * head;
  struct proc_t * tail;
};

struct semaphore_t {
  lock_mode_t mode;      /* free or exclusive */
  spinlock_t  lock;
  pid_t       owner;
  struct wait_queue_t wait_queue;
};
```

Also we add a pointer field into the process descriptor structure (**proc_t**, defined in Section 5.2).

```
struct proc_t {
   .....
  /* add waiting queue list */
  struct proc_t * wait_next;
} proc[NR_PROCS];
```

The pointer links the process descriptor into the waiting queue. The implementations of x-lock is given in Appendix A.3.1.

By using the machine instruction counting method used in Section 5.6, we calculate the average cost for locking. We assume the probability that queue is empty is $\frac{1}{2}$. We avoid other technical details but just give the results.

For **locking**, if the lock is free and the process can get it immediately, the cost is 30 instructions; if the lock is held by another process and the process has to wait, the cost is 134. The **unlocking** takes 62 instructions.

7.3.2 Rw-lock

The second type of lock, rw-lock, takes into account of the classical multiple readers and single writer problem. Its implementation is more complicated than x-lock. The data structures are:

```
#define LOCK_INVALID (-1)
#define LOCK_FREE 0
#define LOCK_S    1
#define LOCK_X    2

typedef int lock_mode_t;

/* the waiting queue */
struct wait_queue_t {
  struct proc_t * head;
  struct proc_t * tail;
};

struct rw_semaphore_t {
  lock_mode_t mode;     /* free, share, or or exclusive */
  spinlock_t  lock;
  int         count;    /* number of holders: 0 free,
                           1 exclusive or shared, >1 shared */
  struct wait_queue_t wait_queue;
};
```

We add three fields to the process descriptor structure (**proc_t** originally defined in Section 5.2).

```
struct proc_t {
   .....
  struct proc_t * wait_next; /* waiting queue link */
  lock_mode_t hold_lock;     /* lock mode the process is holding */
  lock_mode_t wait_for_lock; /* lock mode the process is waiting for */
} proc[NR_PROCS];
```

wait_next is the linking pointer inside the waiting queue. **hold_lock** shows the mode of the lock the process is holding (free, shared, or exclusive). When a process is waiting for a lock in the waiting queue, **wait_for_lock** shows the mode of the lock it is waiting for. The implementations of exclusive lock is given in Appendix A.3.2. Following the same assumption as the above x-lock, we calculate the numbers of instructions of locking and unlocking for rw-locks. For **locking** in shared mode, a successful locking takes 37 instructions, and if the lock cannot be granted, a blocking takes 136 instruction. For **locking** in exclusive mode, the corresponding costs are 32 and 131 respectively. **Unlocking** takes 68 instructions on average.

7.3.3 Comments on semaphore-based lock costs

From the costs obtained from our simplified implementations, we can see that if a locking causes a blocking, the cost is much higher than when it can directly obtain the lock. Another observation is the locking costs of x-locks and rw-locks are quite comparable, although the latter has more complicated logic.

If the data access in memory takes only short time, say, about 10 instructions, using the semaphore-based locking protection can be costly because of the costs of the locking operations themselves. This conforms to the claims in [12] and [4] that semaphores are more suitable for operations which involves I/O and which takes long time.

8 Summary and future work

In the report, we present our step to make an ECP workload. The workload is for further experimental simulation of the transaction processing under different concurrency control protocols. The workload follows the formal model proposed in [14]. The transaction types are transformed from the system calls in the MINIX operating system, and the cost of the states and transitions are measured in terms of CPU instructions. We also take into account the costs of locking operations. We expect to conduct our experiment on this workload alongside the TPC-C workload in [14].

Our future work includes:

1. Because the workload was extracted mainly manually, we need to design the techniques to automate it.

2. We plan to add I/O operations cost to our workload, as they still exist in ECPs.

Appendix

A.1 C programs that handle the selected system calls

In the section, we list the programs of the selected system calls. They are modified from the MINIX source code in [10]. In the programs for **fork** and **exec**, the function **phys_copy** called is a function written in assembly language in MINIX source code.

A.1.1 Constants, types, and data structures

The constants and types are taken from various header files in [10]. We gather their definitions in file **types.h**.

```
/*********************************************************/
/*            constants                                  */
/*********************************************************/
/* error messages */
#define OK   0
#define EGENERIC (-99) /* generic errors */
#define ECHILD (-10)   /* no child process */
#define EAGAIN (-11)   /* resource temporarily unavailable */
#define ENOMEM (-12)   /* no enough memory */

/* memory chunk */
#define CLICK_SIZE 256 /* click unit size */
#define CLICK_SHIFT 8  /* bit shift */

/* segments */
#define NR_SEGS 3   /* segments text, data, stack */
#define T       0   /* text */
#define D       1   /* data */
#define S       2   /* stack */

/* holes */
#define NR_HOLES 128     /* line 18820 */

/* file system */
#define OPEN_MAX 20      /* line 167 */
#define NR_FILPS 128     /* line 19506 */
#define NR_INODES 65     /* line 19507 */

/* processes */
#define MM_PROC_NR 0   /* process id of memory manager */
#define INIT_PID 1     /* process id of init process */
#define IDLE     2     /* idle task */

#define NR_PROCS 32    /* maximum number of user processes */
#define IN_USE 001     /* the process array element is used */

#define WAITING 002   /* the process is waiting: it has called waitpid
                         and is waiting a child to exit */
#define HANGING 004   /* the process is hanging: it has called exit and
                         is waiting for the parent to call waitpid */
#define SEPARATE 040  /* the process has separate text and data segments */

/* misc constants */
#define FALSE 0
#define TRUE  1

#define MAX(a, b) ((a) > (b) ? (a):(b));
#define MIN(a, b) ((a) < (b) ? (a):(b));

#define ARG_MAX  4096  /* args + environ on small
                          machines (for exec()) */
/*********************************************************/
/*           types and structures                        */
/*********************************************************/
typedef int boolean;
typedef int pid_t;            /* process id */
typedef long time_t;   /* time */
typedef unsigned short mode_t; /* file type and permission */
typedef unsighed long off_t;   /* position in a file for r/w */
typedef short dev_t;           /* device id */
typedef unsigned short ino_t;  /* inode id */
typedef unsigned reg_t;        /* register value format */

typedef unsigned int vir_clicks; /* virtual memory chunk */
typedef unsigned int vir_bytes;  /* virtual memory byte */
typedef unsigned int phys_clicks; /* physical memory chunk */
typedef unsigned int phys_bytes; /* physical memory byte */

/* memory segment */
struct mem_map {
  vir_clicks mem_vir;   /* virtual address */
  phys_clicks mem_phys; /* physical address */
  vir_clicks mem_len;   /* length in clicks */
};

/* holes */
struct hole_t {
  phys_clicks h_base;    /* physical starting address */
  phys_clicks h_len;     /* length */
  struct hole_t *h_next; /* pointer to the next entry */
} hole[NR_HOLES];

#define NIL_HOLE (struct hole_t *) 0  /* null hole pointer */

struct hole_t *hole_head;   /* pointer to the first hole */
struct hole_t *free_slots;  /* pointer to the first
                               unused hole table slot */

/* inode */
struct inode_t {
  int    count;             /* reference count:
                               if it is 0, the entry is free */
  time_t i_ctime;           /* when was inode itself changed*/
  dev_t i_dev;              /* which device is the inode on */
  ino_t i_num;              /* inode number on its device */
} inode[NR_INODES];

/* file descriptor */
struct filp_t {
  int filp_count;               /* how many processes share this slot?*/
  struct inode_t *filp_ino;     /* pointer to the inode */
  mode_t mode;                  /* file mode and permission */
  off_t  pos;                   /* position for read or write */
} filp[NR_FILPS];


#define NIL_FILP (struct filp_t *)0  /* null filp pointer */

/* process descriptor */
struct proc_t {
  pid_t  pid;          /* process id */
  pid_t  parent_pid;   /* parent process id */
  pid_t  wpid;         /* pid this process is waiting for */

  char   exitstatus;   /* storage for status when process
                          exits                          */
  char   sigstatus;    /* storage for signal number for
                          killed processes               */

  struct mem_map seg[NR_SEGS]; /* text, data, stack segments */

  /* File identification for sharing */
  ino_t   ino;    /* inode number of exec file */
  dev_t   dev;    /* device number of file system */
  time_t  ctime;  /* inode changed time */

  unsigned proc_flags; /* flag bits */
  reg_t sp;       /* stack pointer */

  struct filp_t *fp_filp[OPEN_MAX]; /* opened file descriptors */
  struct proc_t *p_next_ready;  /* pointer to the next process
                                   in the ready queue          */
} proc[NR_PROCS];

/* ready queues */
#define TASK_Q   0   /* kernel tasks */
#define SERVER_Q 1   /* servers */
#define USER_Q   2   /* user process */
#define NQ 3         /* totally 3 queues */

struct proc_t *rdy_head[NQ]; /* header of a ready queue */
struct proc_t *rdy_tail[NQ]; /* tail of a ready queue */

struct proc_t *proc_ptr; /* current running process */
struct proc_t *bill_ptr; /* process to be billed */
```

A.1.2 Memory allocation

The function **alloc_mem** allocates a memory chunk for for a process segment from the free memory holes, and **free_mem** frees a memory chunk and returns it to the hole list. The **del_slot** is an auxiliary function to delete a hole list slot. They are modified from the functions of the same names in [10].

```
#include "types.h"

phys_clicks alloc_mem(phys_clicks clicks)
{
  register struct hole_t *hp, *prev_ptr;
  phys_clicks old_base;

  hp = hole_head;

  /* scan the hole list */
  while(hp != NIL_HOLE) {
    if(hp->h_len >= clicks) {
      /* we find a hole big enough, use it */
      old_base = hp->h_base;
      hp->h_base += clicks;
      hp->h_len -= clicks;

      if(hp->h_len == 0){
        del_slot(prev_ptr, hp);
      }
      return(old_base);

    }
    prev_ptr = hp;
    hp = hp->h_next;
  }
  return(NO_MEM);
}

void free_mem(phys_clicks base,
              phys_clicks clicks)
{
  boolean flag;
  struct hole_t *hp, *hp1, *prev_hp1, *hp2,
                *new_ptr, *prev_ptr;

  if(clicks == 0) return;

  new_ptr = free_slots;

  /* trying add the freed memory to front of a hole */
  flag = FALSE;
  hp1 = (prev_hp1 = hole_head);
  while(hp1 != NIL_HOLE){
    if(hp1->h_base == base + clicks){
      flag = TRUE;
      hp1->h_base = base;
      hp1->h_len  = hp1->h_len + clicks;
      break;
    }
    prev_hp1 = hp1;
    hp1 = hp1->h_next;
  }

  /* trying add the freed memory to the end of
     a hole */
  hp2 = hole_head;
  while(hp2 != NIL_HOLE){
    if(hp2->h_base + hp2->h_len == base){
      if(!flag){
        flag = TRUE;
        hp2->h_len += clicks;
      }
      else{
        /* we need to merge the two holes */
        hp2->h_len += hp1->h_len;
        del_slot(prev_hp1, hp1);
      }
      break;
    }
```

```
      hp2 = hp2->h_next;
    }

    /* we cannot attach the memory chunk to
       any existing hole. So we have to add
       a new hole to the hole list.         */
    if(!flag){
      new_ptr->h_base = base;
      new_ptr->h_len  = clicks;
      free_slots = new_ptr->h_next;
      hp = hole_head;

      if(hp != NIL_HOLE || base <= hp->h_base) {
        new_ptr->h_next = hp;
        hole_head = new_ptr;
      }
      else{
        while(hp != NIL_HOLE && hp->h_base < base){
          prev_ptr = hp;
          hp = hp->h_next;
        }
        new_ptr->h_next = prev_ptr->h_next;
        prev_ptr->h_next = new_ptr;
      }
    }
  }

  void del_slot(register struct hole_t prev_ptr,
                register struct hole_t hp)
  {
    if(hp==hole_head)
      hole_head = hp->h_next;
    else
      prev_ptr->h_next - hp->h_next;
    hp->h_next = free_slots;
    free_slots = hp;
  }
```

A.1.3 fork

The function is combined and modified from the respective **do_fork** in **MM**, **do_fork** in **FS**, and **do_newmap** in the kernel part in [10].

```
#include <string.h>
#include "types.h"

int fork(pid_t parent_pid){
  int i, j;
  int index;
  pid_t pid;
  boolean flag;
  int proc_in_use;
  struct proc_t parent_proc;
  phys_clicks prog_clicks, child_base = 0;
```

```
phys_bytes  prog_bytes, parent_abs, child_abs;

ino_t ino;
dev_t dev;
time_t ctime;
unsigned proc_flags;
pid_t wpid;
reg_t sp;
struct mem_map text_seg, data_seg, stack_seg;

/* count the number of processes */
proc_in_use = 0;
for(i=0; i<NR_PROCS; i++){
  if(proc[i].proc_flags & IN_USE != 0) {
    proc_in_use++;
  }
}

/* if the number of processes exceeds the
   system limit, return error. */
if (proc_in_use > NR_PROCS) {
  return(EAGAIN);
}

/* get the information of the parent process. */
for(i=0; i<NR_PROCS; i++){
  if((proc[i].proc_flags & IN_USE != 0) &&
     (proc[i].pid == parent_pid))
  {
    parent_proc = proc[i];
    break;
  }
}
ino = parent_proc.ino;
dev = parent_proc.dev;
ctime = parent_proc.ctime;
sp = parent_proc.sp;
wpid = parent_proc.wpid;
proc_flags = parent_proc.proc_flags;

/* calculate the memory that is needed for the new
   process. */
/* stack */
stack_seg.mem_vir = parent_proc.seg[S].mem_vir;
stack_seg.mem_len = parent_proc.seg[S].mem_len;

/* data */
data_seg.mem_vir = parent_proc.seg[D].mem_vir;
data_seg.mem_len = parent_proc.seg[D].mem_len;

/* text */
text_seg.mem_len = parent_proc.seg[T].mem_len;
text_seg.mem_vir = parent_proc.seg[T].mem_vir;

/* total memory size for the new process */
prog_clicks = (phys_clicks)parent_proc.seg[S].mem_len;
prog_clicks += (parent_proc.seg[S].mem_vir - parent_proc.seg[D].mem_vir);

/* allocate memory */
if((child_base = alloc_mem(prog_clicks))==NO_MEM)return(EAGAIN);

/* Create a copy of the parent's core image for the child */
child_abs = (phys_bytes)child_base << CLICK_SHIFT;
parent_abs = (phys_bytes)parent_proc.seg[D].mem_phys << CLICK_SHIFT;

phys_copy(parent_abs, child_abs, prog_bytes);

/* now get an id for the new process */
for(i=0; i<30000; i++){
  flag = FALSE;
  for(j=0; j< NR_PROCS; j++) {
    if((proc[j].pid == i) && (proc[j].proc_flags & IN_USE != 0)){
      flag = TRUE;
      break;
    }
  }
  if(!flag) {
    pid = i;
    break;
  }
}

/* set the text segment physical address */
if(!(proc_flags & SEPARATE)){
  text_seg.mem_phys = child_base;
}
else {
  text_seg.mem_phys = parent_proc.seg[T].mem_phys;
}

/* find an available entry in the . */
for(i=0; i<NR_PROCS; i++){
  if(0 == (proc[i].proc_flags & IN_USE)){
    index = i;
    break;
  }
}

/* there must be one. assign the values to the
   fields */
proc[index].pid = pid;
proc[index].parent_pid = parent_pid;
proc[index].ino = ino;
proc[index].dev = dev;
proc[index].ctime = ctime;
proc[index].sp = sp;
proc[index].seg[T] = text_seg;
proc[index].seg[D] = data_seg;
proc[index].wpid = wpid;
proc[index].proc_flags = proc_flags;

/* share the open file descriptors with the
```

```
     new child */
  for(i=0; i<OPEN_MAX; i++){
    proc[index].fp_filp[i] = parent_proc.fp_filp[i];

    if(proc[index].fp_filp[i] != NIL_FILP){
      (proc[index].fp_filp[i])->filp_count++;
    }
  }

  /* add to ready queue *
  if(rdy_head[USER_Q] == NIL_PROC)
    rdy_tail[USER_Q] = &proc[index];
  rdy_head[USER_Q] -> p_next_ready = rdy_head[USER_Q];
  rdy_head[USER_Q] = &proc[index];

  return (pid);
}
```

A.1.4 exit

The function is combined and modified from the **do_exit** handler in **MM**, and then **do_xit** in the kernel part in [10].

```
#include "types.h"

void do_exit(pid_t pid){
  int i, j;
  struct proc_t *current_proc_ptr, *init_proc_ptr,
         *parent_proc_ptr;
  struct proc_t *xp, *rp, **qtail;
  boolean text_shared, right_child;
  unsigned parent_waiting;

  pid_t parent_pid;
  pid_t wait_pid, parent_wait_pid;
  char exit_status;
  char sig_status;
  unsigned proc_flags, parent_proc_flags;
  ino_t ino;
  dev_t dev;
  time_t ctime;

  /* find the index of the process to exit. */
  for(i=0; i<NR_PROCS; i++){
     if((proc[i].proc_flags & IN_USE != 0) && (proc[i].pid == pid)){
       /* there must be one, because myself exists */
       current_proc_ptr = &proc[i];
       break;
     }
  }

  /* first close all open file. */
  for(i=0; i<OPEN_MAX; i++){
```

```
  if((current_proc_ptr->fp_filp[i])!=NIL_FILP){
    (current_proc_ptr->fp_filp[i])->filp_count--;

    if((current_proc_ptr->fp_filp[i])->filp_count == 0){
      (current_proc_ptr->fp_filp[i])->filp_ino->count--;
    }
    current_proc_ptr->fp_filp[i] = NIL_FILP;
  }
}

/* get all information of the current process. */
parent_pid  = current_proc_ptr->parent_pid;
wait_pid    = current_proc_ptr->wpid;
exit_status = current_proc_ptr->exitstatus;
sig_status  = current_proc_ptr->sigstatus;
proc_flags    = current_proc_ptr->proc_flags;
ino         = current_proc_ptr->ino;
dev         = current_proc_ptr->dev;
ctime       = current_proc_ptr->ctime;

/* remove the process from the ready queue,
   and pick the next process to run. */
if((xp = rdy_head[USER_Q]) == current_proc_ptr){
  rdy_head[USER_Q] = xp->p_next_ready;

  /* pick the next proc to run */
  if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
      proc_ptr = rp;

  }
  else
  if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
     proc_ptr = rp;
  }
  else
  if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
      proc_ptr = rp;
      bill_ptr = rp;

  }else{
  /* No one is ready.  Run the idle task.  The idle task might be made an
   * always-ready user task to avoid this special case.
   */
    bill_ptr = proc_ptr = &proc[IDLE];
  }
}
qtail = &rdy_tail[USER_Q];

while (xp->p_next_ready != current_proc_ptr){
  if ( (xp = xp->p_next_ready) == NIL_PROC) break;
}

if(xp != NIL_PROC){
  xp->p_next_ready = xp->p_next_ready->p_next_ready;
  if (*qtail == current_proc_ptr) *qtail = xp;
}
```

```
/* find whether the text segment is shared */
text_shared = FALSE;
for(i=0; i<NR_PROCS; i++) {
  if((proc[i].proc_flags & (IN_USE | SEPARATE | HANGING))
     != (IN_USE | SEPARATE)) continue;
  if( proc[i].pid == pid || proc[i].ino != ino
     || proc[i].dev != dev || proc[i].ctime != ctime)
    continue;
  text_shared = TRUE;
  break;
}

/* if text is not shared, free the text segment */
if(!text_shared){
  free_mem((current_proc_ptr->seg[T]).mem_phys,
           (current_proc_ptr->seg[T]).mem_len);
}

/* free the data and stack segments */
free_mem((current_proc_ptr->seg[D]).mem_phys,
         (current_proc_ptr->seg[S]).mem_len +
         (current_proc_ptr->seg[S]).mem_vir
         - (current_proc_ptr->seg[D]).mem_vir);

/* find the init process, and let it be parent of the
   children processes if necessary. */
for(i=0; i<NR_PROCS; i++){
  if((proc[i].proc_flags & IN_USE) || (proc[i].pid == INIT_PID)){
    init_proc_ptr = &proc[i];
    break;
  }
}
parent_waiting = init_proc_ptr->proc_flags & WAITING;

/* if parent is waiting */
if(parent_waiting){
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE) &&
      (proc[i].parent_pid == pid)){
      /* if the child already exits */
      if(proc[i].proc_flags & HANGING){
        /* free the child entry. */
        proc[i].proc_flags = 0;
        init_proc_ptr->proc_flags &= ~WAITING;
        break;
      }
    }
  }
}

/* reset the parent of the children to
   the process init */
for(i=0; i<NR_PROCS; i++){
  if((proc[i].proc_flags & IN_USE) &&
     (proc[i].parent_pid == pid))
    {
      proc[i].parent_pid == INIT_PID;
    }
  }

  /* find the parent of the exiting process */
  for(i=0; i<NR_PROCS; i++){
    if((proc[i].proc_flags & IN_USE) &&
       (proc[i].pid = parent_pid)){
      parent_proc_ptr = &proc[i];
      break;
    }
  }
  parent_wait_pid = parent_proc_ptr->wpid;
  parent_wait_pid = parent_proc_ptr->proc_flags & WAITING;

  /* Is the parent waiting for the exiting process? */
  if(parent_wait_pid == -1 || parent_wait_pid == pid){
    right_child = TRUE;
  }
  else{
    right_child = FALSE;
  }

  if(parent_waiting && right_child){
    /* the parent is waiting for the exiting
       process, so free the process descriptor,
       and clear the waiting flag of the parent.
    */
    current_proc_ptr->proc_flags = 0;
    parent_proc_ptr->proc_flags &= ~WAITING;
  }
  else{
    /* set the flag showing the exiting process
       is a zombie.  */
    current_proc_ptr->proc_flags |= HANGING;
  }
}
```

A.1.5 waitpid

The code is modified from the **do_waitpid** function in **MM** in [10].

```
#include "types.h"
int waitpid(pid_t current_pid,
            pid_t pidarg)
{
  int i;
  unsigned parent_proc_flags;
  struct  proc_t  *current_proc, *child_proc;
  pid_t   wait_pid;
  boolean flag;

  /* pidarg == -1, wait for any child,
     pidarg > 0, wait for a special child with the pid specified by
     the parameter.
  */

  if(pidarg != -1 && pidarg <= 0) return(EGENERIC);

  /* get the info of current process. */
  for(i=0; i<NR_PROCS; i++){
    if(proc[i].pid == current_pid){
      current_proc = &proc[i];
      break;
    }
  }

  if(pidarg == -1){
    /* pidarg == -1: waiting for any child. */
    flag = FALSE;
    /* scan the children to see if there is
       a zombie. */
    for(i=0; i<NR_PROCS; i++){
      if((proc[i].proc_flags & IN_USE)
         && proc[i].parent_pid == current_pid){
        child_proc = &proc[i];
        flag = TRUE;
        if(child_proc->proc_flags & HANGING){
          /* find a zombie, delete the process descriptor,
             and clear the parent process flag. */
          child_proc->proc_flags = 0;
          current_proc->proc_flags &= ~WAITING;
          return(OK);
        }
      }
    }
    if(!flag){
      /* Error: we have no child */
      return (ECHILD);
    }
  else{
   /* pidarg is a certain pid
      that we are waiting for */
    flag = FALSE;
    for(i=0; i<NR_PROCS; i++){
      if(proc[i].proc_flags & IN_USE) &&
         proc[i].parent_pid == current_pid &&
         proc[i].pid == pidarg){
        child_proc = &proc[i];
        flag = TRUE;
        if(child_proc->proc_flags & HANGING){
          /* find a zombie, delete the process
             descriptor, and clear the parent
             process flag */
          child_proc->proc_flags = 0;
          current_proc->proc_flags &= ~WAITING;
          return(OK);
        }
      }
    }
    /* no child, return an error. */
    if(!flag){
      /* no qualified children, must be error */
      return(ECHILD);
    }
  }
  current_proc->proc_flags |= WAITING;
  return (OK);
}
```

A.1.6 exec

The code is combined and modified from the respective **do_exec** functions from **MM** and **FS** in [10]. As a simplification, the code here omits the procedure that reads in the executable file from the file system. We assume the image of executable is already in the main memory. The information of the image is passed to the system call handler as parameters.

```
#include "types.h"
#include <string.h>

int exec(pid_t current_pid,
         vir_bytes stack_ptr,
         vir_bytes stk_bytes,
         vir_bytes data_bytes,
         vir_bytes bss_bytes,
         vir_bytes text_bytes,
         vir_bytes tot_bytes,
         ino_t     ino,
         dev_t     dev,
         time_t    ctime,
         int       ft
        )
{
  int i;
  struct proc_t *current_proc_ptr, *mm_proc_ptr;
  vir_bytes src, dst;
  vir_bytes data_vir_addr, data_phy_addr;
  vir_bytes bytes, base, bss_offset, count;
  vir_clicks max_hole_clicks, new_base, proc_clicks;
  boolean text_shared, delete_text;
  vir_clicks text_clicks, data_clicks, gap_clicks, stack_clicks;
  phys_clicks tot_clicks;
  struct hole_t *hp;
  struct mem_map text_seg;
  ino_t old_ino;
  dev_t old_dev;
  time_t old_ctime;
  static char zero[1024];
  static char mbuf[ARG_MAX];

  /* find the process descriptor of the current process */
  for(i=0; i<NR_PROCS; i++){
```

```
  if((proc[i].proc_flags & IN_USE != 0)
     && (proc[i].pid == current_pid)){
    current_proc_ptr = &proc[i];
    old_ino = current_proc_ptr->ino;
    old_dev = current_proc_ptr->dev;
    old_ctime = current_proc_ptr->ctime;
    break;
  }
}

/* store the stacks of the process temporarily at the memory server. */
data_vir_addr = (current_proc_ptr->seg[D]).mem_vir << CLICK_SHIFT;
data_phy_addr = (current_proc_ptr->seg[D]).mem_phys << CLICK_SHIFT;
src = stack_ptr - data_vir_addr + data_phy_addr;

/* find the memory management server. */
for(i=0; i<NR_PROCS; i++){
  if((proc[i].proc_flags & IN_USE != 0)
     && (proc[i].pid == MM_PROC_NR)){
    mm_proc_ptr = &proc[i];
    break;
  }
}

data_vir_addr = (mm_proc_ptr->seg[D]).mem_vir << CLICK_SHIFT;
data_phy_addr = (mm_proc_ptr->seg[D]).mem_phys << CLICK_SHIFT;
dst = (vir_bytes)mbuf - data_vir_addr + data_phy_addr;

phys_copy(src, dst, stk_bytes);

/* find whether the new executable image is already used by
   some existing process and whether we can still share it */
text_shared = FALSE;
for(i=0; i<NR_PROCS; i++) {
  if((proc[i].proc_flags & (IN_USE | SEPARATE | HANGING))
     != (IN_USE | SEPARATE)) continue;
  if( proc[i].pid == current_pid || proc[i].ino != ino
     || proc[i].dev != dev || proc[i].ctime != ctime)
    continue;
  text_shared = TRUE;
  text_bytes = 0;
  text_seg = proc[i].seg[T];  /* shared text */
  break;
}

/* sizes of the segments for the executed proc */
text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
gap_clicks = tot_clicks - data_clicks - stack_clicks;
if ((int)gap_clicks < 0) return(ENOMEM);

/* get the hole of maximum size to assign memory for the
   segments. */
max_hole_clicks = 0;
```

```
hp = hole_head;
while(hp != NIL_HOLE) {
  if(hp->h_len > max_hole_clicks) max_hole_clicks = hp->h_len;
  hp = hp->h_next;
}
if(max_hole_clicks < tot_clicks + text_clicks) return (EAGAIN);

/* if the text is shared by other process, we cannot delete it. */
delete_text = TRUE;
for(i=0; i<NR_PROCS; i++) {
  if((proc[i].proc_flags & (IN_USE | SEPARATE | HANGING))
     != (IN_USE | SEPARATE)) continue;
  if( proc[i].pid == current_pid || proc[i].ino != old_ino
     || proc[i].dev != old_dev || proc[i].ctime != old_ctime)
    continue;
  delete_text = FALSE;  /* it is shared */
  break;
}

/* delete the segments of the original process. */
if(delete_text){
  free_mem((current_proc_ptr->seg[T]).mem_phys,
           (current_proc_ptr->seg[T]).mem_len);
}
free_mem((current_proc_ptr->seg[D]).mem_phys,
         (current_proc_ptr->seg[S]).mem_len +
         (current_proc_ptr->seg[S]).mem_vir
         - (current_proc_ptr->seg[D]).mem_vir);

/* allocate memory for the segments of the executed image */
proc_clicks = tot_clicks + text_clicks;
if((new_base = alloc_mem(proc_clicks))==NO_MEM)return(EAGAIN);

if(!text_shared){
  (current_proc_ptr->seg[T]).mem_phys = new_base;
  (current_proc_ptr->seg[T]).mem_vir = 0;
  (current_proc_ptr->seg[T]).mem_len = text_clicks;
}
else{
  current_proc_ptr->seg[T] = text_seg;
}

/*  data segment. */
(current_proc_ptr->seg[D]).mem_phys = new_base + text_clicks;
(current_proc_ptr->seg[D]).mem_vir = 0;
(current_proc_ptr->seg[D]).mem_len = data_clicks;

/* stack segment. */
(current_proc_ptr->seg[S]).mem_phys = new_base + text_clicks +
                                      data_clicks + gap_clicks;
(current_proc_ptr->seg[S]).mem_vir = data_clicks + gap_clicks;
(current_proc_ptr->seg[S]).mem_len = stack_clicks;

/* "zero" the bbs, gap, and stack */
bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
base = (phys_bytes)((new_base+text_clicks) << CLICK_SHIFT);
```

```
  bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
  base += bss_offset;
  bytes -= bss_offset;
  data_vir_addr = (mm_proc_ptr->seg[D]).mem_vir << CLICK_SHIFT;
  data_phy_addr = (mm_proc_ptr->seg[D]).mem_phys << CLICK_SHIFT;
  src = (vir_bytes)zero - data_vir_addr + data_phy_addr;

  while (bytes > 0) {
    count = MIN(bytes, (phys_bytes) sizeof(zero));
    phys_copy(src, base, count);
    base += count;
    bytes -= count;
  }

  /* copy the stacks back from memory server */
  src = dst;
  dst = (vir_bytes)((current_proc_ptr->seg[S]).mem_phys << CLICK_SHIFT);
  dst += (vir_bytes)((current_proc_ptr->seg[S]).mem_len << CLICK_SHIFT);
  dst -= stk_bytes;
  phys_copy(src, dst, stk_bytes);

  current_proc_ptr->proc_flags &= ~SEPARATE;
  current_proc_ptr->proc_flags |= ft;

  /* set the fields of the process descriptor */
  current_proc_ptr->proc_flags &= ~SEPARATE;
  current_proc_ptr->proc_flags |= ft;
  current_proc_ptr->ino = ino;
  current_proc_ptr->dev = dev;
  current_proc_ptr->ctime = ctime;
  current_proc_ptr->sp  = (reg_t)dst;

  /* close the files open originally by the process before
     exec. */
  for(i=0; i<OPEN_MAX; i++){
    if((current_proc_ptr->fp_filp[i])!=NIL_FILP){
      (current_proc_ptr->fp_filp[i])->filp_count--;

      if((current_proc_ptr->fp_filp[i])->filp_count == 0){
        (current_proc_ptr->fp_filp[i])->filp_ino->count--;
      }
      current_proc_ptr->fp_filp[i] = NIL_FILP;
    }
  }
}
```

A.1.7 brk

The function is modified from the function **do_brk** in **MM** in [10].

```
#include "types.h"
#include "proc.h"

#define DATA_CHANGED 1
```

```
#define STACK_CHANGED 2
int brk(vir_bytes addr, pid_t current_pid){
  int i;
  reg_t sp;
  struct proc_t *current_proc_ptr;
  vir_bytes v;
  vir_clicks new_clicks, sp_clicks, gap_base, lower;
  phys_clicks data_phy, stack_phy;
  vir_clicks data_vir, data_len, stack_vir, stack_len;
  long base_of_stack, delta;
  int changed;

  /* find the current process descriptor and its stack
     pointer. */
  for(i=0; i<NR_PROCS; i++){
      if((proc[i].proc_flags & IN_USE != 0)
         && (proc[i].pid == current_pid)){
      current_proc_ptr = &proc[i];
      sp = current_proc_ptr->sp;
      break;
    }
  }

  v = addr;
  new_clicks = (vir_clicks)( ((long)v + CLICK_SIZE - 1) >> CLICK_SHIFT);

  data_vir = (current_proc_ptr->seg[D]).mem_vir;
  data_phy = (current_proc_ptr->seg[D]).mem_phys;
  data_len = (current_proc_ptr->seg[D]).mem_len;

  /* the address will before the start of data segment. */
  if(new_clicks < data_vir)
    return(ENOMEM);

  new_clicks -= data_vir;

  stack_vir = (current_proc_ptr->seg[S]).mem_vir;
  stack_phy = (current_proc_ptr->seg[S]).mem_phys;
  stack_len = (current_proc_ptr->seg[S]).mem_len;
  base_of_stack = (long) stack_vir + (long) stack_len;

  sp_clicks = sp >> CLICK_SHIFT;        /* click containing sp */
  if (sp_clicks >= base_of_stack) return(ENOMEM); /* sp too high */

  /* Compute size of gap between stack and data segments. */
  delta = (long) stack_vir - (long) sp_clicks;
  lower = (delta > 0 ? sp_clicks : stack_vir);

  /* Add a safety margin for future stack growth. Impossible to do right. */
#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
  gap_base = data_vir + new_clicks + SAFETY_CLICKS;
  if (lower < gap_base) return(ENOMEM); /* data and stack collided */

  /* Update data length (but not data orgin) on behalf of brk() system call. */
  if (new_clicks != data_len) {
    data_len = new_clicks;
    changed |= DATA_CHANGED;
  }

  /* Update stack length and origin due to change in stack pointer. */
  if (delta > 0) {
    stack_vir -= delta;
    stack_phy -= delta;
    stack_len += delta;
    changed |= STACK_CHANGED;
  }

  if(changed & STACK_CHANGED){
    (current_proc_ptr->seg[S]).mem_vir = stack_vir;
    (current_proc_ptr->seg[S]).mem_phys = stack_phy;
    (current_proc_ptr->seg[S]).mem_len = stack_len;
  }

  if(changed & DATA_CHANGED) {
    (current_proc_ptr->seg[D]).mem_vir = data_vir;
    (current_proc_ptr->seg[D]).mem_phys = data_phy;
    (current_proc_ptr->seg[D]).mem_len = data_len ;
  }
}
```

A.2 C/SQL programs for the system calls

A.2.1 fork

```
int fork(int pid){
  EXEC SQL BEGIN DECLARE SECTION;
    int proc_in_use;

    int parent_pid;
    int child_pid;
    int wait_pid;

    int available_holes;
    int hole_base;
    int hole_len;

    int mem_len_stack;
    int mem_len_data;
    int mem_len_text;

    int mem_vir_stack;
    int mem_vir_data;
    int mem_vir_text;

    int mem_phys_data;
    int mem_phys_stack;
    int mem_phys_text;
    int prog_clicks;

    int seg_id_stack;
    int seg_id_data;
    int seg_id_text;

    int curr_proc_flags;

    int exit_status;
    int sig_status;

    int ino;
    int dev;
    char change_time[27];
    int filp_id;

    int stack_pointer;
  EXEC SQL END DECLARE SECTION;

  int i;
  int child_base;
  int child_abs;
  int parent_abs;
  int ret_value;

  /* wether we reach the upperlimit of number of processes? */
  EXEC SQL
  SELECT count(*) into :proc_in_use
  FROM proc;

  if (proc_in_use >= NR_PROCS) {
    ret_value = EAGAIN;
    goto out;
  }

  /* select all fields from the proc */
  parent_id = pid;
  EXEC SQL
  SELECT data_segment into :seg_id_data,
         stack_segment into :seg_id_stack,
         text_segment into :seg_id_stack,
         ino    into :ino,
         dev    into :dev,
         change_time into :change_time,
         proc_flags into :curr_proc_flags,
         wait_pid into :wait_pid,
         stack_pointer into :stack_pointer
  FROM proc
  WHERE proc.pid = :parent_id;

  /* calculate the memory that is needed for the new
     process */
  /* stack */
  EXEC SQL
  SELECT segment.length into :mem_len_stack,
         segment.virtual_address into :mem_vir_stack
```

```
FROM  segment
AND   segment.seg_id = :seg_id_stack;

/* data */
EXEC SQL
SELECT segment.virtual_address into :mem_vir_data,
       segment.physical_address into :mem_phys_data
FROM segment
WHERE  segment.seg_id = :seg_id_data;

/* calculate the full memory */
/* for text, it is either shared (in same physical
   memory) by multiple processes, or merged with
   data segment, so we handle it later. */
mem_len_data = mem_vir_stack - mem_vir_data
proc_click = mem_len_stack + mem_len_data;

/* get memory chunk from the holes and adjust the hole size */
EXEC SQL DECLARE hole_cursor CURSOR FOR
  SELECT len, base
  FROM hole
  WHERE len >= :proc_click
FOR UPDATE;

EXEC SQL OPEN hole_cursor;

EXEC SQL FETCH hole_cursor into :hole_len, :hole_base;

if(SQL_CODE != 0){
  EXEC CLOSE hole_cursor;
  ret_value = EGAGAIN;
  goto err_label;
}

child_base = hole_base;
hole_len -= proc_click;
hole_base += proc_click;

if(hole_len>0){
  EXEC SQL
  UPDATE hole SET len = :hole_len, base = :hole_base
  WHERE CURRENT OF CURSOR hole_cursor;
}
else{
  /* if the segment takes the entire hole, the hole is
   deleted */
  EXEC SQL
  DELETE FROM hole
  WHERE CURRENT OF CURSOR hole_cursor;
}

EXEC SQL CLOSE hole_cursor;

/* copy the content of main memory */
child_abs = child_base << CLICK_SHIFT;
parent_abs = mem_phys_data << CLICK_SHIFT;
```

```
phys_copy(parent_abs, child_abs, prog_bytes);

/* now get a child id */
/* it must be different from any existing process */
EXEC SQL DECLARE pid_cursor FOR
SELECT pid
FROM pids
WHERE NOT EXISTS
  (SELECT *
   FROM proc
   WHERE proc.pid = pids.pid
  );

EXEC OPEN pid_cursor;
EXEC fetch pid_cursor into :child_pid;
EXEC close pid_cursor;

/* set the address of the child's segments */
mem_phys_stack = childbase + mem_len_data;
mem_phys_data = childbase;

/* get segment ids */
/* times 10 to get room for text segment id */
/* overwrite the value of the variables, which are the parent's segments,
   except the text segment if it shared. */
seg_id_data = mem_phys_data;
seg_id_stack = mem_phys_stack;
seg_id_data *= 10;
seg_id_stack*= 10 + 2;

/* insert into segment table */
EXEC SQL
INSERT INTO segment
VALUES(:seg_id_stack, :mem_vir_stack, :mem_phys_stack, :mem_len_stack);

EXEC SQL
INSERT INTO segment
VALUES(:seg_id_data, :mem_vir_data, :mem_phys_data, :mem_len_data);

/* if non-separate text and data */
if(!(curr_proc_flags & SEPARATE)){
  /* make the segment id */
  seg_id_text =  seg_id_data + 1;

  /* text segment and data segment are not separated */
  EXEC SQL
  INSERT INTO segment
  VALUES(:seg_id_text, :mem_vir_text, :mem_phys_data, :mem_len_text);
}
/* else share the original text segment */

/* insert a row into the process table */
EXEC SQL
INSERT INTO
  proc(pid,
       parent_id,
           wait_pid,
           exit_status,
           sig_status,
           data_segment,
           text_segment,
           stack_segment,
           ino,
           dev,
           change_time,
           stack_pointer,
           proc_flags)
    VALUES(:child_pid, :parent_pid, :wait_pid,
           :exit_status, :sig_status,
           :seg_id_data, :seg_id_text, :seg_id_stack,
           :ino, :dev, :change_time,
           :stack_pointer, :curr_proc_flags);

  /* share the files with the child */
  EXEC SQL DECLARE proc_filp_cursor CURSOR FOR
    SELECT filp_id
    FROM proc_filp
    WHERE proc_id = :parent_pid;

  EXEC SQL OPEN proc_filp_cursor;

  if(SQLCODE == 0){
    while(true){
     EXEC SQL FETCH proc_filp_cursor into :filp_id;
     if(SQLCODE != 0 ) break;

     EXEC SQL
     INSERT INTO proc_filp
     VALUES(:child_pid, :filp_id);

     EXEC SQL
     UPDATE filp SET filp_count = filp_count + 1
     WHERE filp_id = :filp_id;
    }
  }

  EXEC SQL CLOSE proc_filp_cursor;

  /* add to ready queue */
  EXEC SQL
    INSERT INTO ready_user_proc
    VALUES (:child_pid);

out:
  return (ret_value);
}
```

A.2.2 exit

```
exit(int pid) {
```

```
EXEC SQL BEGIN DECLARE SECTION;
  int current_pid;
  int parent_pid;
  int wait_pid;

  int exit_status;
  int sig_status;

  int data_segment;
  int text_segment;
  int stack_segment;

  int proc_flags;

  int file_count;
  int text_share_count;
  int generic_count;

  int mem_vir_text;
  int mem_phy_text;
  int mem_len_text;

  int mem_vir_data;
  int mem_phy_data;
  int mem_len_data;

  int mem_vir_stack;
  int mem_phy_stack;
  int mem_len_stack;

  int mem_phy_base;
  int mem_phy_len;
  int mem_phy_end;

  int parent_wait_pid;
  int parent_proc_flags;

  int init_proc_pid;
  int child_id;
  int child_proc_flags;

  int ino;
  int dev;
  int ctime;

  int base1;
  int len1;

  int base2;
  int len2;

  int temp_pid;
  int temp_proc_flags;

  int temp_ino;
  int temp_dev;
  int temp_c_time;
EXEC SQL BEGIN DECLARE SECTION;

int flag;
int right_child;
int parent_waiting;
boolean delete_text;
int temp_len;

current_pid = pid;

/* first close all open files */
/* we simplify file close to delete */
/* decrement the reference count in filp */
EXEC SQL UPDATE filp set filp_count = filp_count - 1
WHERE EXISTS (
  SELECT * FROM proc_filp
  WHERE proc_filp.proc_id = :current_pid
  AND   proc_filp.filp_id = filp.filp_id
);

/* decrement the count of the inode in memory of an
   open file */
EXEC SQL UPDATE inode set count = count - 1
WHERE EXISTS (
  SELECT * from filp
  WHERE filp.filp_count = 0
  AND filp.inode_id = inode.inode
)

/* delete from the proc-filp relationship */
EXEC SQL DELETE FROM proc_filp
WHERE proc_id = :current_pid;

/* delete the entries which has no
   process that references it */
EXEC SQL DELETE FROM filp
WHERE filp_count = 0;

/* delete the entries of inodes which has
   been closed by all processes */
EXEC  SQL DELETE FROM inode
WHERE count = 0;

/* get all info of the exiting process */
EXEC SQL SELECT
  parent_pid into :parent_pid,
  wait_pid into :wait_pid,
  exit_status into :exit_status,
  sig_status into :sig_status,
  data_segment into :data_segment,
  text_segment into :text_segment,
  stack_segment into :stack_segment,
  proc_flags into :proc_flags,
  ino into :ino,
  dev into :dev,
  ctime into :ctime
FROM proc
WHERE pid = :current_pid;

/* get the process out of the ready queue */
DELETE FROM ready_user_proc
WHERE pid = :current_pid

/* delete segments */
delete_text = FALSE;

EXEC SQL DECLARE proc_cursor CURSOR FOR
  SELECT pid, ino, dev, ctime, proc_flags
  FROM proc;

EXEC OPEN proc_cursor;

while(SQLCODE == 0){
  EXEC FETCH proc_cursor
  into :temp_pid, :temp_ino, :temp_dev, :temp_ctime,
       :temp_proc_flags;

  if( (temp_proc_flags & (SEPARATE | HANGING)) !=
       SEPARATE ) continue;
  if((temp_pid == pid) || (temp_ino != ino) ||
     (temp_dev != dev) || (temp_ctime != ctime))
    continue;
  else{
    delete_text = TRUE;
    break;
  }
}
EXEC CLOSE proc_cursor;

/* check the case of shared text segment */
if(delete_text){
  /* get the physical address and length of the text */
  EXEC SQL SELECT physical_address into :mem_phy_text,
                  length into :mem_len_text
  FROM segment
  WHERE segment.sid = :text_segment;

  /* next, we free the text_segment and insert the
     physical segment into the hole table */

  /* fist see whether we can merge it to other holes. */
  flag = FALSE;

  /* now we have the end of the text segment */
  mem_phy_end = mem_phy_text + mem_len_text;

  /* can we merge it with the hole after it */
  EXEC SQL DECLARE hole_cursor1 CURSOR FOR
    SELECT base, len
    FROM hole
    WHERE base = :mem_phy_end
```

```
FOR UPDATE;

EXEC SQL OPEN hole_cursor1;
EXEC SQL FETCH hole_cursor1 INTO :base1, :len1;

if(SQLCODE == 0){
  /* we can merge */
  flag = TRUE;
  len1 = len1 + mem_len_text;
  base1 = mem_phy_text;

  /* merge */
  EXEC SQL UPDATE hole
  SET base = :base1,
      len  = :len1
  WHERE current of hole_cursor1;
}

/* can we merge it with the hole ahead of it */
EXEC SQL DECLARE hole_cursor_2 FOR
  SELECT base, len
  FROM hole
  WHERE base + len = :mem_phy_text
FOR UPDATE;

EXEC SQL OPEN hole_cursor_2;
EXEC SQL FETCH hole_cursor_2 to :base2, :len2;

/* we can merge */
if(SQLCODE == 0){
  /* Have we already merged it with the one afterwards. */
  if(!flag){
    /* no. so merge it to the one in front of it. */
    flag = TRUE;

    EXEC SQL UPDATE hole
    SET len = :len2 + :mem_len_text
    WHERE current of hole_cursor_2;
  }
  else{
    /* the hole afterwards is from the merging in
       the above step. */
    /* merge the hole with the hole after it. */
    EXEC SQL UPDATE hole
    SET len = :len2 + :len1
    WHERE current of hole_cursor_2;

    EXEC SQL DELETE FROM hole
    WHERE current of hole_cursor_1;
  }
}

EXEC SQL CLOSE hole_cursor_1;
EXEC SQL CLOSE hole_cursor_2;

if(!flag){
    /* no merge at all, add the segment to the hole table. */
    EXEC SQL INSERT INTO hole
    VALUES(:mem_phy_text, :mem_len_text);
  }
}

/* handle the stack and data */
/* they are contiguous physical segments */
/* so we return them together */
EXEC SQL SELECT mem_phy_data into :mem_phy_data,
                mem_vir_data into :mem_vir_data
FROM proc, segment
WHERE segment.seg_id = :data_segment
AND proc.pid = :current_pid;

EXEC SQL SELECT mem_vir_stack into :mem_vir_stack,
                mem_len_stack into :mem_len_stack
FROM segment
WHERE segment.seg_id = :stack_segment

temp_len = mem_vir_stack -
           mem_vir_data + mem_len_stack;
mem_phy_end = mem_phy_data + temp_len;

flag = FALSE;

/* merge with the one after it */
EXEC SQL DECLARE CURSOR hole_cursor_3 FOR
  SELECT base, len
  FROM hole
  WHERE base = :mem_phy_end
FOR UPDATE;

EXEC SQL OPEN hole_cursor_3;
EXEC SQL FETCH hole_cursor_3 INTO :base1, :len1;

if(SQLCODE = 0){
  flag = TRUE;
  base1 = mem_phy_data;
  len1 = len1 + temp_len;

  EXEC SQL UPDATE hole
  SET base = :base1,
      len  = :len1
  WHERE current of hole_cursor_3;
}

/* merge the one ahead of it */
EXEC SQL DECLARE CURSOR hole_cursor_4 FOR
SELECT base, len
FROM hole
WHERE base + len = :mem_phy_data;
FOR UPDATE;

EXEC SQL OPEN hole_cursor_4;
EXEC SQL OPEN hole_cursor_4 into :base2, :len2;
```

```
/* has the segment already merged with the one
   behind it? */
if(SQLCODE == 0){
  if(!flag){
    flag = TRUE;
    len2 += temp_len;

    EXEC SQL UPDATE hole
    SET len = :len2
    WHERE current of hole_cursor_4;
  }
  else{
    /* it may cause merge of three segments */

    EXEC SQL UPDATE hole
    SET len = :len1 + :len2
    WHERE current of hole_cursor_4;

    EXEC SQL DELETE FROM hole
    WHERE current of hole_cursor_3;
  }
}

EXEC SQL CLOSE hole_cursor_3;
EXEC SQL CLOSE hole_cursor_4;

if(!flag){
  /* no merge at all, so insert the hole
     into the table. */
  mem_phy_len = temp_len;
  EXEC SQL INSERT INTO hole
  VALUES(:mem_phy_data, :mem_phy_len);
}

/* set the segments pointers to NULL */
/* this is because they are foreign keys */
/* then we will delete the three segments in
   the segment table */
EXEC SQL UPDATE proc
         SET stack_segment = NULL,
             text_segment = NULL,
             data_segment = NULL;

/* delete the segments in the segment table */
EXEC SQL DELETE FROM segment
WHERE seg_id = :stack_segment
   OR seg_id = :data_segment;

if(delete_text){
  EXEC SQL DELETE FROM segment
  WHERE seg_id = :text_segment;
}

/* Let INIT_PROC adopt my children */
EXEC SQL DECLARE CURSOR init_cursor FOR
```

```
SELECT proc_flags
FROM proc
WHERE pid = :init_proc_pid
FOR UPDATE;

EXEC SQL OPEN init_cursor;
EXEC FETCH init_cursor into :parent_proc_flags;

parent_waiting = parent_proc_flags & WAITING;

/* if parent is waiting */
if(parent_waiting){
  EXEC SQL DECLARE child_proc_cursor CURSOR FOR
    SELECT proc_flags FROM proc
    WHERE  parent_pid = :current_pid
  FOR UPDATE;

  EXEC SQL OPEN child_proc_cursor;

  while(SQLCODE == 0){
    EXEC SQL FETCH child_proc_cursor into :child_proc_flags;

    if(child_proc_flags & HANGING){
      /* delete the child if the parent is waiting and the child
         has already exited */
      DELETE FROM proc
      WHERE CURRENT OF child_proc_cursor;

      parent_proc_flags &= ~WAITING;

      /* clear the waiting flag */
      EXEC SQL UPDATE proc
           SET proc_flags = :parent_proc_flags;
      WHERE current of init_cursor;

      break;
    }
  }

  EXEC SQL CLOSE init_cursor;
  EXEC SQL CLOSE child_proc_cursor;
}

EXEC SQL UPDATE proc
SET parent_pid = :init_proc_id
WHERE parent_pid = :current_pid;

/* Is my parent waiting? */
EXEC SQL DECLARE CURSOR parent_cursor FOR
SELECT wait_pid into :parent_wait_pid,
                proc_flags   into :parent_proc_flags
FROM proc
WHERE proc.pid = :parent_pid;
FOR UPDATE;

EXEC SQL OPEN parent_cursor;
```

```
  EXEC SQL FETCH parent_cursor INTO :parent_wait_pid,
                                    :parent_proc_flags;

  parent_waiting = parent_proc_flags & WAITING;

  /* wait for any child or wait for me? */
  if(parent_wait_pid == -1 || parent_wait_pid == current_pid){
    right_child = TRUE;
  }
  else{
    right_child = FALSE;
  }

  if(parent_waiting && right_child){
    /* parent is waiting for me */
    /* delete the table record for the process */
    DELETE FROM proc
    WHERE pid = :current_pid;

    parent_proc_flags &= ~WAITING;

    /* clear the waiting flag */
    EXEC SQL UPDATE proc
         SET proc_flags =: parent_proc_flags
    WHERE CURRENT OF parent_cursor;
  }
  else{
    /* parent not waiting, set the HANGING flag */
    proc_flags |= HANGING;

    EXEC SQL UPDATE proc
         SET proc_flags = :proc_flags
    WHERE CURRENT OF parent_cursor;
    /* the entry will be deleted when the parent calls wait */
  }
  EXEC CLOSE parent_cursor;
}
```

A.2.3 waitpid

```
int waitpid(int, pid, int pidarg)
/* pidarg == -1, wait for any child,
   pidarg > 0,   wait for a special pid */
{
  EXEC SQL BEGIN DECLARE SECTION;
    int current_pid;
    int wait_pid;

    int child_proc_flags;
    int parent_proc_flags;
    int wait_pid;

    int child_count;
  EXEC SQL END DECLARE SECTION;
```

```
if(pidarg != -1 && pidarg <= 0) return (ERROR);

wait_pid = pidarg;

/* get the parent. */
current = pid;
EXEC SQL DECLARE CURSOR parent_flag_cursor FOR
SELECT proc_flags
FROM proc
WHERE pid = :current_pid
FOR UPDATE;

EXEC OPEN parent_flag_cursor;
EXEC FETCH parent_flag_cursor into :parent_proc_flags;

if(pidarg == -1){
  /* wait for any child */
  /* get a children */
  EXEC SQL DECLARE child_proc_cursor FOR
    SELECT proc_flags
    FROM proc
    WHERE parent_pid = :current_pid
  FOR UPDATE;

  EXEC SQL OPEN child_proc_cursor;

  if(SQLCODE != 0) {
    /* Error in getting children */
    return(ECHILD);
  }

  while(SQLCODE == 0){
    EXEC SQL FETCH child_proc_cursor into :child_proc_flags;

    if(child_proc_flags & HANGING){
      /* delete the child if the parent is waiting and the child
         has already exited */
      DELETE FROM proc
      WHERE CURRENT OF child_proc_cursor;

      parent_proc_flags &= ~WAITING;

      /* clear the waiting flag. */
      EXEC SQL UPDATE proc
      SET proc_flags := parent_proc_flags
      WHERE CURRENT OF parent_flag_cursor;

      EXEC SQL CLOSE child_proc_cursor;
      EXEC SQL CLOSE parent_flag_cursor;
      return(OK);
    }
  }
}
else{  /* pidarg is wait_pid */
  /* wait for a certain pid */
```

```
    /* get the children */
    EXEC SQL DECLARE child_proc_cursor FOR
      SELECT proc_flags
      FROM proc
      WHERE parent_pid = :current_pid
      AND  pid = :wait_pid;
    FOR UPDATE;

    EXEC SQL OPEN child_proc_cursor;

    if(SQLCODE != 0){
      /* no qualified children, must be an error */
      return(ECHILD);
    }

    EXEC SQL FETCH child_proc_cursor into :child_proc_flags;

    if(child_proc_flags & HANGING){
      /* delete the child if the parent is waiting and the child
         has already exited */
      DELETE FROM proc
      WHERE CURRENT OF child_proc_cursor;

      parent_proc_flags &= ~WAITING;

      /* clear the waiting flag. */
      EXEC SQL UPDATE proc
      SET proc_flags := parent_proc_flags
      WHERE CURRENT OF parent_flag_cursor;

      EXEC SQL CLOSE child_proc_cursor;
      EXEC SQL CLOSE parent_flag_cursor;
      return(OK);
    }
  }

  /* we have qualified children, but can not find a child that has
     exited */
  /* set the parent's status to waiting */
  parent_proc_flags |= WAITING;

  EXEC SQL UPDATE proc
  SET proc_flags := parent_proc_flags
  WHERE CURRENT OF parent_flag_cursor;

  EXEC SQL CLOSE child_proc_cursor;
  EXEC SQL CLOSE parent_flag_cursor;

  return (OK);
}
```

A.2.4 exec

```
#define MM_PROC_NR 0
```

```
#define ARG_MAX  4096

into exec(int pid,
          int stack_ptr,
          int stk_bytes,
          int data_bytes,
          int bss_bytes,
          int text_bytes,
          int tot_bytes,
          int ino_param,
          int dev_param,
          int ctime_param,
          int ft) {
  EXEC SQL BEGIN DECLARE SECTION;
    int current_pid;

    int mm_proc_pid;

    int stack_segment_phy;
    int stack_segment_vir;
    int stack_segment_len;

    int data_segment_phy;
    int data_segment_vir;
    int data_segment_len;

    int text_segment_phy;
    int text_segment_vir;
    int text_segment_len;

    int segment_end;

    int text_count;

    int text_seg_id;
    int data_seg_id;
    int stack_seg_id;

    int old_text_seg_id;
    int old_data_seg_id;
    int old_stack_seg_id;

    int proc_flags;

    int max_hole_size;

    int hole_base;
    int hole_len;

    int ino;
    int dev;
    int ctime;

    int old_ino;
    int old_dev;
    int old_ctime;
    int temp_pid;
    int temp_ino;
    int temp_dev;
    int temp_ctime;

    int proc_clicks;
    int count_generic;

    int stack_pointer;

    int base1;
    int len1;

    int base2;
    int len2;
  EXEC SQL END DECLARE SECTION;

  char mbuf[ARG_MAX];
  int  src, dst;
  boolean text_shared;
  boolean delete_old_text;
  boolean flag;
  int new_base;
  int text_clicks, data_clicks, stack_clicks,
      total_clicks, gap_clicks;
  int base, bytes, count, bss_offset;
  static char zero[1024];
  int mm_data_phy, mm_data_vir;
  int temp_len;
  int ft;

  ino = ino_param;
  dev = dev_param;
  ctime = ctime_param;

  /* copy the content of the stack to the mem space of mm task */
  current_pid = pid;
  EXEC SQL SELECT
    text_segment into :old_text_seg_id,
    stack_segment into :old_stack_seg_id,
    data_segment into :old_data_seg_id,
    proc_flags  into :proc_flags,
    ino     into :old_ino,
    dev     into :old_dev,
    ctime   into :old_ctime
  FROM proc
  WHERE proc_id = :current_pid;

  EXEC SQL SELECT segment.physical_address into :data_segment_phy,
                  segment.virtual_address into :data_segment_vir
  FROM segment
  WHERE seg_id = old_data_seg_id;

  mm_data_vir = data_segment_vir << CLICK_SHIFT;
  mm_data_phy = data_segment_phy << CLICK_SHIFT;
```

```
src = stack_ptr - mm_data_vir + mm_data_phy;

mm_proc_pid = MM_PROC_NR;
EXEC SQL SELECT segment.physical_address into :data_segment_phy,
                segment.virtual_address into :data_segment_vir
FROM proc, segment
WHERE segment.seg_id = proc.data_segment
AND proc_id = :mm_proc_pid;

mm_data_vir = data_segment_vir << CLICK_SHIFT;
mm_data_phy = data_segment_phy << CLICK_SHIFT;
dst = mbuf - mm_data_vir + mm_data_phy;

phys_copy(src, dst, stack_bytes);

/* find text share  */
text_shared = FALSE;

EXEC SQL DECLARE proc_cursor CURSOR FOR
SELECT pid, ino, dev, ctime, proc_flags, text_segment
FROM proc;

EXEC OPEN proc_cursor;

while(SQLCODE == 0){
  EXEC FETCH proc_cursor
  into :temp_pid, :temp_ino, :temp_dev, :temp_ctime,
       :temp_proc_flags, :text_seg_id;

  if( (temp_proc_flags & (SEPARATE | HANGING)) !=
       SEPARATE ) continue;
  if((temp_pid == current_pid) || (temp_ino != ino) ||
     (temp_dev != dev) || (temp_ctime != ctime))
    continue;
  else{
    text_shared = TRUE;
    text_bytes = 0;
    break;
  }
}
EXEC CLOSE proc_cursor;

/* new segments for the executed proc */
text_clicks = ((unsigned long) text_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
data_clicks = (data_bytes + bss_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
stack_clicks = (stk_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
tot_clicks = (tot_bytes + CLICK_SIZE - 1) >> CLICK_SHIFT;
gap_clicks = tot_clicks - data_clicks - stack_clicks;
if ( (int) gap_clicks < 0) return(ENOMEM);

/* whether there is a hole that can fit the memory */
EXEC SQL SELECT max(len) into :max_hole_size
FROM hole;

if(max_hole_size < tot_clicks + text_clicks) return (EAGAIN);
```

```
/* first release the old segment */
/* text segment */
delete_old_text = TRUE;

EXEC SQL DECLARE proc_cursor CURSOR FOR
  SELECT pid, ino, dev, ctime, proc_flags
  FROM proc;

EXEC OPEN proc_cursor;

while(SQLCODE == 0){
  EXEC FETCH proc_cursor
  into :temp_pid, :temp_ino, :temp_dev, :temp_ctime,
       :temp_proc_flags;

  if( (temp_proc_flags & (SEPARATE | HANGING)) !=
       SEPARATE ) continue;
  if((temp_pid == current_pid) || (temp_ino != old_ino) ||
     (temp_dev != old_dev) || (temp_ctime != old_ctime))
    continue;
  else{
    delete_old_text = FALSE;  /* it is shared, cannot delete it */
    break;
  }
}
EXEC CLOSE proc_cursor;

/* check the case of shared text segment. */
if(delete_old_text){
  /* get the physical address and length of the text */
  EXEC SQL SELECT physical_address into :text_segment_phy,
                  length into :text_segment_len
  FROM segment
  WHERE segment.sid = :old_text_seg_id;

  /* next, we free the text_segment and insert the
     physical segment into the hole table. */
  /* fist see whether we can merge it to other holes. */
  flag = FALSE;

  /* now we have the end of the text segment. */
  segment_end = text_segment_phy + text_segment_len;
  /* can we merge it with the hole after it? */
  EXEC SQL DECLARE hole_cursor1 CURSOR FOR
    SELECT base, len
    FROM hole
    WHERE base = :segment_end
  FOR UPDATE;

  EXEC SQL OPEN hole_cursor1;

  EXEC SQL FETCH hole_cursor1 INTO :base1, :len1;

  if(SQLCODE == 0){
    /* we can merge */
    flag = TRUE;
    len1 = len1 + text_segment_len;
    base1 = text_segment_phy;

    /* merge */
    EXEC SQL UPDATE hole
    SET base = :base1,
        len  = :len1
    WHERE current of hole_cursor1;
  }

  /* can we merge it with the hole ahead of it? */
  EXEC SQL DECLARE hole_cursor_2 FOR
    SELECT base, len
    FROM hole
    WHERE base + len = :text_segment_phy
  FOR UPDATE;

  EXEC SQL OPEN hole_cursor_2;
  EXEC SQL FETCH hole_cursor_2 to :base2, :len2;

  /* we can merge */
  if(SQLCODE == 0){
    /* Have we already merged it with the one afterwards. */
    if(!flag){
      /* no. so merge it to the one in front of it. */
      flag = TRUE;
      EXEC SQL UPDATE hole
      SET len = :len2 + :text_segment_len
      WHERE current of hole_cursor_2;
    }
    else{
      /* the hole afterwards is from the merging in
         the above step. */
      /* merge the hole with the hole after it. */
      EXEC SQL UPDATE hole
      SET len = :len2 + :len1
      WHERE current of hole_cursor_2;

      EXEC SQL DELETE FROM hole
      WHERE current of hole_cursor_1;
    }
  }

  EXEC SQL CLOSE hole_cursor_1;
  EXEC SQL CLOSE hole_cursor_2;

  if(!flag){
    /* no merge at all, add the segment to the hole table. */
    EXEC SQL INSERT INTO hole
    VALUES(:text_segment_phy, :text_segment_len);
  }
}

/* handle the stack and data */
/* they are contiguous physical segments */
/* so we return them together */
EXEC SQL SELECT mem_phy_data into :data_segment_phy,
                mem_vir_data into :data_segment_vir
FROM segment
WHERE segment.seg_id = :old_data_seg_id;

EXEC SQL SELECT mem_vir_stack into :stack_segment_vir,
                mem_len_stack into :stack_segment_len
FROM segment
WHERE segment.seg_id = :old_stack_seg_id;

temp_len = stack_segment_vir - data_segment_vir
           + stack_segment_len;
segment_end = data_segment_phy + temp_len;

flag = FALSE;

/* merge with the one after it */
EXEC SQL DECLARE CURSOR hole_cursor_3 FOR
  SELECT base, len
  FROM hole
  WHERE base = :segment_end
FOR UPDATE;

EXEC SQL OPEN hole_cursor_3;

EXEC SQL FETCH hole_cursor_3 INTO :base1, :len1;

if(SQLCODE = 0){
  flag = TRUE;
  base1 = data_segment_phy;
  len1 = len1 + temp_len;

  EXEC SQL UPDATE hole
  SET base = :base1,
      len  = :len1
  WHERE current of hole_cursor_3;
}

/* merge the one ahead of it */
EXEC SQL DECLARE CURSOR hole_cursor_4 FOR
SELECT base2, len2
FROM hole
WHERE base + len = :segment_data_phy;
FOR UPDATE;

EXEC SQL OPEN hole_cursor_4;
EXEC SQL OPEN hole_cursor_4 into :base2, :len2;

/* has the segment already merged with the one
   behind it? */
if(SQLCODE == 0){
  if(!flag){
    flag = TRUE;
    len2 += temp_len;
    EXEC SQL UPDATE hole
    SET len = :len2
    WHERE current of hole_cursor_4;
  }
  else{
    /* it may cause merge of three segments */
    EXEC SQL UPDATE hole
    SET len = :len1 + :len2
    WHERE current of hole_cursor_4;

    EXEC SQL DELETE FROM hole
    WHERE current of hole_cursor_3;
  }
}

EXEC SQL CLOSE hole_cursor_3;
EXEC SQL CLOSE hole_cursor_4;

if(!flag){
  len1 = temp_len;

  EXEC SQL INSERT INTO hole
  VALUES(:mem_phy_data, :len1);
}

/* set the segments pointers to NULL */
/* this is because they are foreigh keys */
/* then we will delete the three segments in
   the segment table */
EXEC SQL UPDATE proc
         SET stack_segment = NULL,
             text_segment = NULL,
             data_segment = NULL
WHERE proc_id = :current_pid;

/* delete the segments in the segment table */
EXEC SQL DELETE FROM segment
WHERE seg_id = :old_stack_seg_id
   OR seg_id = :old_data_seg_id;

if(delete_old_text){
  EXEC SQL DELETE FROM segment
  WHERE seg_id = :old_text_seg_id;
}

proc_clicks = tot_clicks + text_clicks;

/* get memory chunk from the holes and adjust the hole size */
EXEC SQL DECLARE hole_cursor CURSOR FOR
  SELECT len, base
  FROM hole
  WHERE len >= :proc_clicks
FOR UPDATE;

EXEC SQL OPEN hole_cursor;
EXEC SQL FETCH hole_cursor into :hole_len, :hole_base;
```

```
new_base = hole_base;
hole_len -= proc_click;
hole_base += proc_click;

if(hole_len > 0){
  EXEC SQL
  UPDATE hole SET len = :hole_len, base = :hole_base
  WHERE CURRENT OF CURSOR hole_cursor;
}
else{
  /* if the segment takes the entire hole, the hole is
   deleted */
  EXEC SQL
  DELETE FROM hole
  WHERE CURRENT OF CURSOR hole_cursor;
}

EXEC SQL CLOSE hole_cursor;

/* if the text is not shared yet, insert a segment for it. */
if(!text_shared){
  text_segment_phy = new_base;
  text_segment_vir = 0;
  text_segment_length = text_clicks;
  text_seg_id = new_base * 10 + 1;

  EXEC SQL INSERT INTO segment
  (seg_id, virtual_address, physical_address, length)
  VALUES(:text_seg_id, :text_segment_vir, :text_segment_phy,
         :text_segment_length);
}

data_segment_phy = new_base + text_clicks;
data_segment_vir = 0;
data_segment_length = data_clicks;
data_seg_id = data_segment_phy * 10;

EXEC SQL INSERT INTO segment
(seg_id, virtual_address, physical_address, length)
VALUES(:data_seg_id, :data_segment_vir, :data_segment_phy,
       :data_segment_length);

stack_segment_phy = new_base + text_clicks + data_clicks + gap_clicks;
stack_segment_vir = data_clicks + gap_clicks;
stack_segment_length = stack_clicks;
stack_seg_id = stack_segment_phy * 10 + 2;

EXEC SQL INSERT INTO segment
(seg_id, virtual_address, physical_address, length)
VALUES(:stack_seg_id, :stack_segment_vir, :stack_segment_phy,
       :stack_segment_length);

/* zero the segments in memory */
bytes = (phys_bytes)(data_clicks + gap_clicks + stack_clicks) << CLICK_SHIFT;
base = (phys_bytes)data_segment_phy << CLICK_SHIFT;
```

```
bss_offset = (data_bytes >> CLICK_SHIFT) << CLICK_SHIFT;
base += bss_offset;
bytes -= bss_offset;

while (bytes > 0) {
  count = MIN(bytes, (phys_bytes) sizeof(zero));
  sys_copy(ABS, 0, zero - mm_data_vir + mm_data_phy, ABS, 0, base, count);
  base += count;
  bytes -= count;
}

/* recovery the stacks from the MM server. */
src = dst;
dst = stack_segment_phy << CLICK_SHIFT;
dst += stack_segment_len << CLICK_SHIFT;
dst -= stk_bytes;
stack_pointer = dst;

phys_copy(src, dst, stk_bytes);

proc_flags &= ~SEPARATE;
proc_flags |= ft;

/* update the process descriptor. */
EXEC SQL UPDATE proc
SET text_segment = :text_seg_id,
    data_segment = :data_seg_id,
    stack_segment = :stack_seg_id
    ino = :ino,
    ctime = :ctime,
    dev = :dev,
    stack_pointer := stack_pointer;
    proc_flags = :proc_flags,
WHERE proc.pid = :current_pid;

/* close open files of the process before calling exec */
EXEC SQL UPDATE filp set filp_count = filp_count - 1
WHERE EXISTS (
  SELECT * FROM proc_filp
  WHERE proc_filp.proc_id = :current_pid
  AND   proc_filp.filp_id = filp.filp_id
);

/* decrement the count of the inode in memory of an
   open file */
EXEC SQL UPDATE inode set count = count - 1
WHERE EXISTS (
  SELECT * from filp
  WHERE filp.filp_count = 0
  AND filp.inode_id = inode.inode
)

/* delete from the proc-filp relationship */
EXEC SQL DELETE FROM proc_filp
WHERE proc_id = :current_pid;
```

```
  /* The above update filp may changes
     filp_count to 0
  */
  /* delete the entries which has no
     process that references it */
  EXEC SQL DELETE FROM filp
  WHERE filp_count = 0;

  / * delete the inode if no process is
      referencing to it */
  EXEC SQL DELETE FROM inode
  WHERE count = 0;
}
```

A.2.5 brk

```
#define DATA_CHANGED 1
#define STACK_CHANGED 2

brk(int addr, int pid){
  EXEC SQL BEGIN DECLARE SECTION;
    int current_pid;

    int stack_pointer;

    int data_phy;
    int data_vir;
    int data_len;
    int data_seg;

    int stack_phy;
    int stack_vir;
    int stack_len;

    int stack_seg;
  EXEC SQL BEGIN DECLARE SECTION;

  vir_bytes v, new_sp;
  vir_clicks new_clicks, sp_click, gap_base, lower;

  long base_of_stack, delta;
  int changed;

  /* get the segment info of the process. */
  current_pid = pid;
  EXEC SQL SELECT
    data_segment into :data_seg,
    stack_segment into :stack_seg,
    stack_pointer into :stack_pointer
  FROM proc
  WHERE proc.pid = :current_pid;

  v = addr;
  new_clicks = (vir_clicks) ( ((long)v + CLICK_SIZE - 1) >> CLICK_SHIFT);
  EXEC SQL SELECT
    virtual_address into :data_vir,
    physical_address into :data_phy,
    length into :data_len
  FROM segment
  WHERE proc.data_segment = :data_seg;

  if(new_clicks < data_vir)
    return(ENOMEM);

  new_clicks -= data_vir;

  EXEC SQL SELECT
    virtual_address into :stack_vir,
    physical_address into :stack_phy,
    length into stack_len
  FROM segment
  WHERE proc.stack_segment = :stack_seg;

  base_of_stack = (long) stack_vir + (long) stack_len;
  sp_clicks = stack_pointer >> CLICK_SHIFT; /* click containing sp */
  if (sp_clicks >= base_of_stack) return(ENOMEM);   /* sp too high */

  /* compute size of gap between stack and data segments. */
  delta = (long) stack_vir - (long) sp_clicks;
  lower = (delta > 0 ? sp_clicks : stack_vir);

#define SAFETY_BYTES  (384 * sizeof(char *))
#define SAFETY_CLICKS ((SAFETY_BYTES + CLICK_SIZE - 1) / CLICK_SIZE)
  gap_base = data_vir + new_clicks + SAFETY_CLICKS;
  if (lower < gap_base) return(ENOMEM); /* data and stack collided */

  /* update data length (but not data origin) on behalf of brk() system call. */
  if (new_clicks != data_len) {
    data_len = new_clicks;
    changed |= DATA_CHANGED;
  }

  /* Update stack length and origin due to change in stack pointer. */
  if (delta > 0) {
    stack_vir -= delta;
    stack_phy -= delta;
    stack_len += delta;
    changed |= STACK_CHANGED;
  }

  if(changed & STACK_CHANGED){
    EXEC SQL UPDATE segment
      SET virtual_address = :stack_vir,
          physical_address = :stack_phy,
          length = :stack_len
    WHERE seg_id = :stack_seg;
  }

  if(changed &  DATA_CHANGED) {
    EXEC SQL UPDATE segment
      SET length = :data_len
    WHERE seg_id = :data_seg;
  }

  return(OK);
}
```

A.3 Implementation of semaphore-based locks

A.3.1 X-lock

```
#define INVALID_PID (-1000)

/* add the process to the tail of the waiting queue */
void add_wait(struct wait_queue_t *wait_queue, struct proc_t *proc)
{
  proc->wait_next = NIL_PROC;

  /* add to tail */
  if(wait_queue->tail){
    (wait_queue->tail)->wait_next = proc;
  }
  wait_queue->tail = proc;

  /* add to header if the queue is empty */
  if(!(wait_queue->head)){
    (wait_queue)->head = proc;
  }
}

/* remove the process from the ready list,
   and schedule another process to run. */
void unschedule(struct proc_t * current_proc_ptr){
  struct proc_t *xp, *rp, **qtail;
    /* copied from unready() and pick_proc() */
  if((xp = rdy_head[USER_Q]) == current_proc_ptr){
    rdy_head[USER_Q] = xp->p_next_ready;

    /* pick the next proc to run */
    if ( (rp = rdy_head[TASK_Q]) != NIL_PROC) {
        proc_ptr = rp;
    }
    else
    if ( (rp = rdy_head[SERVER_Q]) != NIL_PROC) {
       proc_ptr = rp;
    }
    else
    if ( (rp = rdy_head[USER_Q]) != NIL_PROC) {
        proc_ptr = rp;
        bill_ptr = rp;
    }else{
    /* No one is ready.  Run the idle task.  The idle task might be made an
```

```
     * always-ready user task to avoid this special case.
     */
      bill_ptr = proc_ptr = &proc[IDLE];
    }
  }
  qtail = &rdy_tail[USER_Q];

  while (xp->p_next_ready != current_proc_ptr){
    if ( (xp = xp->p_next_ready) == NIL_PROC) break;
  }

  if(xp != NIL_PROC){
    xp->p_next_ready = xp->p_next_ready->p_next_ready;
    if (*qtail == current_proc_ptr) *qtail = xp;
  }
}

/* acquire an exclusive lock. */
void x_lock(struct proc_t* current_proc, struct semaphore_t * sem){
  spin_lock(&(sem->lock));

  if(sem->mode != LOCK_X) {
     sem->mode = LOCK_X;
     sem->owner = current_proc->pid;
     spin_unlock(&(sem->lock));
     return;
  }
  else{
     /* must be locked */

     /* if the current proc already locks it, do nothing. */
     if(sem->owner == current_proc->pid){
       spin_unlock(&(sem->lock));
       return;
     }

     /* block it. */
     add_wait(&(sem->wait_queue),current_proc);
     unschedule(current_proc);
     spin_unlock(&(sem->lock));
  }
}

/* remove the process from the head of the waiting queue */
void remove_wait(struct wait_queue_t *queue, struct proc_t *proc){
  proc = NIL_PROC;
  if(queue->head != NIL_PROC){
    proc = queue->head;
    queue->head = (queue->head)->wait_next;
  }
}

/* put the process to the ready list */
void schedule(struct proc_t *current_proc){
  if(rdy_head[USER_Q] == NIL_PROC)
    rdy_tail[USER_Q] = current_proc;
  rdy_head[USER_Q] -> p_next_ready = rdy_head[USER_Q];
  rdy_head[USER_Q] = current_proc;
}

/* unlock */
void x_unlock(struct proc_t *current_proc, struct semaphore_t *sem){
  struct proc_t *proc_wakeup;

  spin_lock(&(sem->lock));
  /* I am not the owner of the lock. do nothing. */
  if(sem->mode != LOCK_X || sem->owner != current_proc->pid){
    spin_unlock(&(sem->lock));
    return;
  }

  /* wake up the first proc in the wait queue. */
  remove_wait(&(sem->wait_queue), proc_wakeup);
  if(NIL_PROC != proc_wakeup){
    /* if wait queue is not empty */
    schedule(proc_wakeup);
    sem->owner =  proc_wakeup->pid;
  }
  else{
    /* no waiting process */
    sem->owner = INVALID_PID;
    sem->mode = LOCK_FREE;
  }

  spin_unlock(&(sem->lock));
}
```

A.3.2 Rw-lock

The function **schedule**, **unschedule**, **add_wait**, and **remove_wait** are the same as in Appendix A.3.1. The other functions added are:

```
/* check what kind of lock the first process in the wait queue
   requests for */
lock_mode_t get_header_wait_for(struct wait_queue_t *queue){
  if(queue->head == NIL_PROC){
    return LOCK_INVALID;
  }
  else{
    return queue->head->wait_for_lock;
  }
}

/* locking */
void rw_lock(struct proc_t         *current_proc,
             struct rw_semaphore_t *sem,
             lock_mode_t            mode)
{
  spin_lock(&(sem->lock));
  if(sem->mode == LOCK_FREE) {
     /* lock free, so obtain it. */
     sem->mode = mode;
     sem->count = 1;
     current_proc->hold_lock = mode;
     spin_unlock(&(sem->lock));
     return;
  }
  else{
     /*
     /* if the current proc already locks it, do nothing. */
     if(mode <= current_proc->hold_lock){
       spin_unlock(&(sem->lock));
       return;
     }
     else
     if(mode == LOCK_S && sem->mode == LOCK_S &&
        (sem->wait_queue).head == NIL_PROC)
     {
       sem->mode = mode;
       sem->count++;
       current_proc->hold_lock = mode;
       spin_unlock(&(sem->lock));
       return;
     }

     /* block it. */
     add_wait(&(sem->wait_queue),current_proc);
     current_proc->wait_for_lock = mode;
     unschedule(current_proc);
     spin_unlock(&(sem->lock));
  }
}

/* unlocking */
void rw_unlock(struct proc_t *current_proc,
               struct rw_semaphore_t *sem){
  struct proc_t *proc_wakeup;
  spin_lock(&(sem->lock));
  /* I am not the owner of the lock. do nothing. */
  if(sem->mode == LOCK_FREE ||
     sem->mode != current_proc->hold_lock) {
    spin_unlock(&(sem->lock));
    return;
  }

  sem->count--;

  /* wake up procs in the wait queue. */
  if(sem->count == 0){
    remove_wait(&(sem->wait_queue), proc_wakeup);

    /* the wait queue is empty? */
    if(NIL_PROC != proc_wakeup){
      /* not empty, wake up the first one */
      sem->mode = proc_wakeup->wait_for_lock;
```

```
        proc_wakeup->hold_lock = sem->mode;
        schedule(proc_wakeup);

        /* wake up more readers if necessary */
        if(LOCK_S == sem->mode){
          while(get_header_wait_for(&(sem->wait_queue)) == LOCK_S)
          {
            remove_wait(&(sem->wait_queue), proc_wakeup);
            proc_wakeup->hold_lock = LOCK_S;
            schedule(proc_wakeup);
          }
        }
      }
      else{
        /* wait queue empty */
        sem->mode = LOCK_FREE;
      }
    }
    current_proc->hold_lock = LOCK_FREE;
    spin_unlock(&(sem->lock));
  }
```

References

[1] Linux Cross-Reference. URL http://lxr.linux.no.

[2] Transaction Processing Performance Council. URL http://www.tpc.org.

[3] Blott, S., and Korth, H. F. An almost-serial protocol for transaction execution in main-memory database systems. In *Proceedings of the 28th VLDB Conference* (2002).

[4] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel*, 2nd ed. O'Reilly, 2002.

[5] Chan, P. K. C. Optimizing OQL on Legacy Main Memory Data Structures with Existential Graphs. Master's thesis, University of Waterloo, Waterloo, ON., Canada, 1997.

[6] Garcia-Molina, H., and Salem, K. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering 4*, 2 (December 1992), 509–516.

[7] Lai, S. Y. M. On integrating legacy main memory data structure with database schema. Master's thesis, University of Waterloo, Waterloo, ON., Canada, 1999.

[8] Liu, H., Toman, D., and Weddell, G. Fine Grained Information Integration with Description Logics. In *Working Notes of the 2002 International Description Logics Workshop (DL-2002)* (Toulouse, France, 2002), p. to appear.

[9] STANCHEV, L., AND WEDDELL, G. Index Selection for Compiled Database Applications in Embedded Control Programs. In *Proceedings of CASCON 2002* (Toronto, Canada, September - October 2002), pp. 156–170.

[10] TANENBAUM, A. S., AND WOODHULL, A. S. *Operating Systems: design and implementation*, 2nd ed. Prentice Hall, 1997.

[11] TOMAN, D., AND WEDDELL, G. E. Query Processing in Embedded Control Programs. In *Proceedings of the Second International Workshop on Databases in Telecommunications* (2001), no. 2209 in Lecture Notes in Computer Science, Springer-Verlag, pp. 68–87.

[12] VAHALIA, U. *Unix Internals: the New Frontiers.* Prentice Hall, 1996.

[13] WEIKUM, G., AND VOSSEN, G. *Transactional Information Systems: Theory, Algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann, 2001.

[14] YU, H., AND WEDDELL, G. Investigation in Tree Locking for Compiled Database Applications. In *Proceedings of CASCON 2004* (Toronto, Canada, October 2004), pp. 217–231.